Department of Computer Science

Bachelor Thesis

Minimal implementation of the OCI runtime specification

Implementing an experimental subset of the OCI runtime specification to reduce overhead for embedded Linux systems

Abgabedatum:

26. August 2024

Eingereicht von:
github.com/m4schini

Referent:
Prof. Dr. Ulrich Hoffmann
Fachhochschule Wedel

Betreut von:
Dane-Marvin Dahlem
MaibornWolff GmbH

Contents

List of Figures

List of Tables

List of Acronyms

LXC Linux Container

CLI Command Line Interface

API Application Programming Interface

IDL Interface Definition Language

OCI Open Container Initiative

PID process id

NIS Network Information System

IPC Interprocess Communication

RPC Remote Procedure Call

UTS Unix Timesharing system

PaaS Platform as a Service

fd file descriptor

symlink Symbolic link

cgroup control group

protobuf protocol buffer

confs container filesystem

statedir container state directory

syscall Linux system call

roci Reduced OCI Runtime

Abstract

This thesis explores the intersection of cloud technologies and embedded systems by investigating the potential of adapting containerization to resource-constrained embedded systems. With the rise of IoT and edge computing, there is an increasing opportunity to bring the portability and automation benefits of containers into embedded deployments without overwhelming the limited resources available.

The primary objectives were to gain a deep understanding of the Linux features that enable containerization, especially namespaces and cgroups, and to develop a minimal container runtime compatible with the Open Container Initiative (OCI) specifications. The research involved analyzing existing container technologies, designing a minimal runtime, and implementing it in a way that preserves key container benefits while minimizing overhead.

Despite significant challenges, due to the lack of standardized APIs and the exploratory nature of implementing the OCI runtime specification, the thesis successfully demonstrates the feasibility of creating a minimal container runtime. Although a big part of the OCI specification including network isolation was excluded due to its complexity, the resulting runtime is compatible with container managers like Podman and can create containers similar to standard implementations. This work is an exploration into the adaptation of cloud-native technologies in embedded systems, covering an interesting intersection of modern software engineering.

1

Introduction

1.1. Motivation

Containers are currently one of the most popular forms of deployment in cloud environments. Over the last decades, they completely changed the server and data center landscape. Moving large parts of the industry away from heavyweight virtual machine isolation towards lightweight software isolation with containers. Containers and container orchestration software like Kubernetes are the fundamental building blocks that made the *Cloud* possible, mainly through the high degree of portability and automation that containers provide.

But for the most part containers have remained on developer machines and in the *Cloud*, far away from environments like embedded systems. However, with the rise of IoT and edge computing, more general-purpose hardware and operating systems such as Linux are finding their way into embedded systems. It has become apparent that this hardware is being pushed to its limits simply by using these tools which were designed for systems with very different resource budgets compared to the low-powered hardware found in embedded systems.

This thesis wants to provide additional insights into the intersection of cloud technologies and embedded systems. Building an experimental foundation for the adaptation and adoption of cloud technologies into a new environment with new rules.

The central question discussed in this thesis is, how much of the overhead can be stripped away from OCI containers while preserving their portability and automation possibilities.

1.2. Goal

This thesis has three primary goals:

1. Explaining the Linux features that make containerization possible

2. Building a container that provides the portability and automation capabilities of OCI containers

3. Implementing an Open Container Initiative (OCI) specification compatible container runtime. *Compatible* means that the interoperability with tools that depend on the OCI specification should be preserved but full compliance with the specification is out of scope.

1.3. Overview

Including this introduction, this thesis consists of seven chapters. Starting with a comprehensive theoretical background in chapter two that covers a short history of containerization technologies, the Linux features that make containerization possible, including an in depth section about namespaces and namespace types as these will be the most important components for the implementation of the custom runtime and a short summary of the OCI specifications. Chapter three will provide a short overview over three different approaches to containerization on embedded systems. Following these theoretical explanations the practical work on the custom implementation of the reduced feature set of the OCI runtime specification will be described. Chapter four discusses the design of the implementation. It will extend on the theoretical background of the OCI runtime specification and discuss the features that are either going to be implemented or not, and the expected compromises that result from of trying to implement a compatible but not compliant OCI runtime. Chapter five will provide specifics about the actual implementation, covering decisions on how the specification is implemented in code, what additional libraries are being used, and of the new runtime and its restrictions. After which the runtime will be validated and benchmarked in chapter six to provide a basic comparison to other OCI runtimes. Chapter seven will conclude the thesis, with a summary, a potential outlook and an additional note regarding the availability and licensing of the custom OCI implementation.

2

Theoretical Background

This chapter explains the fundamental concepts, features and software needed to understand their use in this Thesis. Different container technologies and a selection of their history will be explored.

2.1. Container

Container technology on its own and as part of Kubernetes has reached global adoption with end-users across sectors. It is now the most common technology to package, deploy and manage software applications in cloud environments. [TC24]

The umbrella term *Container technology* covers a huge catalogue of different software, specifications and processes. **Containers** are a standardized unit of software that enables portability and automated handling of the application lifecycle. This thesis will focus on the common Linux application container, often called a *Docker container*, as defined by the Open Container Initiative (OCI) specifications (cf. 2.3).

The term ***Container*** is often used interchangeably with the term ***Container Image***, but they describe two different things. The Container Image is a bundle of the software with all its dependencies (cf. 2.3.2) and a container can be created from this image with a *container runtime* (cf. 2.3.1).

Furthermore there must be made a distinction between a container manager (also known as container engine) and a container runtime. A **container manager** is a high level component that provides more features than just the management of a containerized application. Popular container manager like Docker and Podman provide user interfaces and also implement building, distribution and downloads of container images as described in 2.3.2 and 2.3.3.

The **container runtime** is a low level component that is responsible for the management of the container lifecycle, from creation to deletion.

This architecture and terminology is common in the OCI ecosystem and can also be found outside of it, but in- and outside of the ecosystem these terms may be used interchangeable or with a completely different meaning.

The next subsections describe some of the historic step stones that lead to the OCI container.

2.1.1. LXC

Linux Container (LXC) is one of the earliest userspace[1] tools for containers. However the concept of containerization is older and has its roots in `chroot` (cf. 2.2.1).

The development of LXC started in 2008 [HR15] and used several Linux features that existed already for some time but matured and were not used in this configuration before. The most important features are namespaces (cf. 2.2.3) and control groups (cgroups) (cf. 2.2.5). These are also important in later implementations of containerization like Docker. [LXC24]

LXC was primarly developed to provide *system containers*, which serve a similar role to virtual machines. Since the release of 3.0.0, LXC can also provides application containers (cf. 2.1.2) and is compatible with OCI containers. [Gra18]

2.1.2. Docker

Docker was introduced by Solomon Hykes during the PyCon conference in March 2013 [Hyk13] as part of DotCloud, a Platform as a Service (PaaS) company. The release of Docker marked a significant advancement in container technology, making it accessible and straightforward for developers to use.

Similar to LXC, Docker leverages existing Linux isolation features like namespaces and cgroups, to provide lightweight and portable containers. Docker has experienced rapid adoption within the developer community. When Docker was released there

[1] User space refers to all of the code in an operating system that lives outside of the kernel [RM15]

were important differences to LXC, one of them being that the *Docker container* is an application container, not a system container like the ones provided by LXC during that time. Application containers have a more narrow focus than system container and are commonly used to run only a single process, and almost always a very small amount of processes inside of them.

The initial release of Docker in 2013 was a monolithic daemon taking care of the complete container lifecycle as well as container and image management. In 2015 *Docker Inc.* started to break the Docker monolith down into multiple smaller parts [HD15], releasing runc (cf. 2.3.1), a lightweight component only responsible for the lifecycle management of a container.

Today the software named *Docker* is a high-level container management tool that relies on Containerd[2] for its containerization functionality. Containerd is using OCI compliant container runtimes, by default runc, to manage the lifecycle of containers.

2.1.3. Open Container Initiative

In July 2015, the Open Container Initiative (OCI) was announced, initially under the name Open Container Project [Ope15]. The Open Container Initiative was founded by Docker and other industry leaders at the time to create an open governance structure, for the purpose of creating standards around container formats and runtimes.

For that purpose Docker Inc. donated it's current implementation of runc and their specifications for Containers and Images to the OCI in 2015. [Ope15]

Based on this donated reference implementation and specification, the three OCI specifications were created (see 2.3).

2.1.4. Podman

Podman is a high-level container manager, which was developed after the OCI was established, and directly based on the OCI specifications. It started as the libpod package in the CRI-O project under the Kubernetes umbrella and became a independent tool.

[2] Containerd describes itself as a industry-standard container runtime [cT24]

It has less restrictive licensing, then Docker and has become a popular alternative to Docker. Like Docker is using Containerd, Podman is using conmon to interact with the OCI runtime, as such it is categorized as an indirect *runtime caller* (cf. 4.3.1).

2.2. Linux

Linux is a open-source unix-like[3] kernel and also a generic name for the family of operating systems that are based on the Linux kernel. Linux or so called Linux distributions[4] are widely used in different environments like datacenters, smartphones, embedded systems and traditional personal computers. While operating systems like Windows and MacOS have the highest market share on personal computers, Linux has the server market dominance with $\sim 97\%$ of the top one million web servers running on Linux based operating systems. [W3C15].

2.2.1. Syscalls

Linux system calls (syscalls) build the fundamental interface between an application in the userspace and the Linux kernel. These syscalls can be called directly but most of them have wrapper functions defined in the *Standard C library* and alternative implementations based on the *Standard C library*. [Lin24q]

Syscalls provide the interface to most of the common Linux features, like *change dir* (`cd`), (`chroot`) and more complex features like creating new processes with the `clone` syscall.

A full list of the available syscalls can be found in the Linux Manual page `syscalls(2)` [Lin24q]. Listed here is a selection of syscalls used in the custom implementation of the OCI runtime specification in Chapter 5.

[3] *unix-like* refers to operating systems that behave similar to UNIX but that are not necessarily conforming to a UNIX specification

[4] So called distros are operating systems that are based on the linux kernel and include a specific collection of software, system tools and drivers

mount

All file descriptors on Unix and Unix-like systems are arranged in one big file hierarchy rooted at `/`. The mount command can be used to attach additional filesystems in the root file hierarchy. These filesystems can be filesystems on other devices or virtual filesystems like `procfs` (cf. 2.2.2). [Lin24i]

clone

Using the syscall clone or fork creates a duplication of the calling process. The new process is referred to as *child process* and the calling process is referred to as the *parent process*. [Lin24e]

The clone syscall creates a new child process and provides precise control over what pieces of execution context are shared between the parent and child process. This includes creating the child process inside new namespaces. The use of `clone` to create namespaces is described in 2.2.3. [Lin24e]

setsid

This syscall creates a new session and process group for the calling process. This can be used to detach a new process from its parent. After the child process was added to a new session it will not be effected by its former parent, instead continuing to run even after the former parent process stopped. [Lin24e]

unshare

Similar to clone where a new process can be spawned inside new namespaces, unshare can be used to change the execution context of the calling process. [Lin24s]

Unshare does have more limitations than clone. For example when unsharing the pid namespace (cf. 2.2.4), it does not change the PID of the calling process to 1, but the first child process of the calling process will be inside the new namespace with PID 1. [Lin24s]

mkfifo

This syscall creates a file descriptor (fd)[5] for a pipe.

A pipe is a unidirectional data channel used for Interprocess Communication (IPC). It is a low level Linux component that streams bytes from the writing process to the reading process. Anonymous pipes are often used in the terminal to stream the data output of one command into another, using the character “|”.

The syscall `mkfifo` can be used to create a **named pipe**. Named pipes behave very similar to anonymous pipes but they provide a fd as an interface. The writing process opens the fd in `NamedPipe` mode and with writing permissions and the reading process opens the fd with `NamedPipe` mode and reading permissions. When only one process has opened the fd, all operations block and only unblock as soon as both processes have opened the fd. [Lin24h]

exec

The family of `exec` syscalls effectively replaces the calling process with a new process. It does this by replacing the current *process image*[6] with a different process image. After the replacement the new process image has the same process id as the process image that was replaced. [Lin24d]

chroot

In linux two pointers are associated with a process at all times, the current directory and the root directory. By default the root directory is pointed to the root of the filesystem, if the process opens a file starting with ‘/‘, it will resolve the path in relation to the root pointer of the process. Not the actual root of the filesystem. [Bel23].

The `chroot` syscall sets that root directory pointer to a different directory. After the root pointer is changed all absolute paths resolved in the process will be resolved in

[5] File descriptors are a reference to a metadata information block in the kernel that represents a file or something that implements the interface of a file. fds are used as the API for a lot of io operations. [Tay11]

[6] The term *Process image* describes a executable loaded into memory, either already running or ready to be executed[Flo97]

relation to the new root. [Lin24c]

2.2.2. procfs

On Unix-like systems active processes are tracked in a *virtual filesystem* (commonly also referred to as *pseudo-filesystem*) called **procfs**. This section will explain the parts of this virtual filesystem that are relevant for the implementation in Chapter 5.

Inside procfs

Each paragraph describes a directory or file from a selection of the files and directories inside procfs.

/proc/<pid> Each subdirectory that is identified by the process id (PID) of a process contains files and subdirectories containing information about that process [Lin24o]. **PID** is the acronym for process id, which is a numeric identifier assigned to a process when it is created. This identifier is unique and can be used to distinguish between processes that have the same human readable name.

Some of the files and directories inside a `<pid>` directory are:

- `/uid_map`, `/gid_map` Contains ID mappings for User namespaces (cf. 2.2.4)
- `/mounts`, `/mountinfo`, `/mountstats` Provide views of mounts visible to the process. Controlled by Mount namespaces (cf. 2.2.4)

/proc/<pid>/ns/ Each process has a `ns` subdirectory that contains one entry for each namespace that is supported by the syscall `setns`. Opening one of the files inside this directory will return a file descriptor for the corresponding namespace. The namespace remains alive at least as long as the file descriptor is kept open. The files appear as symbolic links and contain the namespace type and *inode number*[7]. [Lin24l]

[7] Each file has an inode containing metadata about the file and the inode number is a unique id of the file [Lin24f]

/proc/<pid>/fd/ This subdirectory contains one entry for each file that is open in the process. Each entry is a symbolic link to the actual file.

By default three open fds can be expected, “0” linking to standard input, “1” linking to standard output and “2” linking to standard error.

/proc/self When a process accesses this symbolic link, it resolves to the process’s own `/proc/<pid>` directory. [Lin24o]

Virtual filesystems like procfs are userspace interfaces to the kernel and can not be used like most filesystems. Procfs can be mounted anywhere, but by convention it is expected under `/proc` [LT15a].

2.2.3. Namespaces

A namespace is an abstraction of a Linux system resource that can be used to isolate that system resource for the processes inside the namespace. Changes to the system resource in the namespace are only visible to processes inside but not outside the namespace. [Lin24l]

Creating Namespaces

Namespaces are created using the syscalls `unshare` (cf. 2.2.1) and `clone` (cf. 2.2.1). The `clone` syscall is used to create a new process and when provided with new-namespace flags (i.e. flags prefixed with `CLONE_NEW`) the child process will be a member of those namespaces.

```
1 pid_t pid = clone(child_func, stack, CLONE_NEWNS | SIGCHLD, arg);
```

Figure 2.1.: Example of a `clone` syscall using the *standard c library*

In figure 2.1, child_func is the function executed by the child process, stack is the stack space for the child process, `CLONE_NEWNS` specifies the creation of a new mount namespace (cf. 2.2.4), and `SIGCHLD` indicates the signal sent when the child process terminates.

Assigning Namespaces

Processes can be assigned to already existing namespaces with the `setns` syscall. This syscall needs the fd referring to the namespace (cf. 2.2.2) and then assigns the calling process to that namespace. The file descriptors referring to namespaces can be found in the `\ns` subdirectory of a process directory in `procfs` (cf. 2.2.2).

```
1 int fd = open("/proc/<pid>/ns/net", O_RDONLY);
2 setns(fd, CLONE_NEWNET);
```

Figure 2.2.: Example of a `setns` syscall using the *standard c library*

In 2.2, the file descriptor "`fd`" is associated with the network namespace (cf. 2.2.4) of a process identified by `<pid>`, and `setns` moves the calling process into that network namespace.

Removing Namespaces

A namespace is removed or "freed" when the last process within it is terminated. However, this is not always the case, as additional conditions may need to be met before the namespace can be freed. According to the Linux manual pages [Lin24l], the additional conditions include:

- An open file descriptor or a bind mount exists for the corresponding `/proc/<pid>/ns/*` file.
- The namespace is hierarchical (i.e., a PID or user namespace), and has a child namespace.
- It is a user namespace that owns one or more nonuser namespaces.
- It is a PID namespace, and there is a process that refers to the namespace via a `/proc/<pid>/ns/pid_for_children` symbolic link.
- It is a time namespace, and there is a process that refers to the namespace via a `/proc/<pid>/ns/time_for_children` symbolic link.
- It is an IPC namespace, and a corresponding mount of an mqueue filesystem (see [Lin24k]) refers to this namespace.

- It is a PID namespace, and a corresponding mount of a procfs (cf. 2.2.2) refers to this namespace.

2.2.4. Namespace Types

The namespace types available in Linux are: Cgroup, IPC, Network, Mount, PID, Time, User and UTS. This is a selection of namespaces used to construct containers:

PID namespace

This namespace isolates PID assignments, for example when creating a new PID namespace and running a new process inside, this process can have the PID 1. But while this process has PID 1 inside the namespace it will have a different PID in the *initial namespace*[8] [Iva17].

The process with the PID 1 is a *init process*. In the initial namespace this process runs from start to shutdown of the system and is responsible for managing all daemon processes on Linux. When a new PID namespace is created the *init process* is the first process started in that namespace. If that process is terminated it will also terminate all other processes in the namespace, by sending a SIGKILL signal. [Bou22]

The **PID namespace** provides isolation of the PID number space and enables the duplicate assignment of PIDs over multiple PID namespaces. It is also possible to nest PID namespaces. Since Linux 3.7 this is possible to a depth of 32 layers [Lin24l]. In each layer the process has a unique PID assigned, there are no guarantees that the PID will be the same as in an *ancestor namespace*[9]. This is a one-way relation, ancestor namespaces can see all processes in *decending namespaces*[10] but descending namespaces can not see the processes of ancestor namespaces. [Lin24n]

`CLONE_NEWPID` A new PID namespace can be created with the syscalls `clone` (2.2.1) or `unshare` (2.2.1) using the `CLONE_NEWPID` flag.

[8] *initial namespace* refers to the default/root namespace that all namespaces decent from
[9] *ancestor namespace* refers to all namespaces from the parent namespace to the root namespace
[10] *descending namespace* refers to all namespaces from a direct child namespace and all namespaces created from it's children and so on.

```c
// Create a new pid namespace
if (unshare(CLONE_NEWPID) == -1) {
    return EXIT_FAILURE;
}
```

Figure 2.3.: Example of creating a new PID namespace using the *standard c library*

Because a lot of Linux software is dependent on the conventional filesystem hierarchy [LT15b], the Linux manual pages recommend when creating a new PID namespace to also change the root directory and mount `procfs` at `/proc` in relation to the new root [Lin24l].

Network namespace

Network namespaces isolate networking related system resources like network devices, IPv4 and IPv6 protocol stacks, IP routing table, firewall rules the `/proc/net` directory, the `/sys/class/net` directory, various files under `/proc/sys/net`, port numbers (sockets), etc. Also the Unix sockets used for inter-process communication are isolated with the network namespace and not the IPC namespace. [Lin24m] [Iva17]

A physical network device can only exist in one network namespace. When a network namespace is freed, all physical network devices associated with the namespace are returned to the initial network namespace [Lin24m]. Physical network devices can be accessed across namespaces using a virtual network device pair *veth*. Using this abstractions tunnels can be created between network namespaces, which then enables inter-namespace access to the physical network device. When a namespace is freed all contained virtual network device pairs are destroyed [Lin24m].

While this provides the basis of virtual networking that tools like Docker provide, it is to note that Container manager often use virtual switches and network tooling like Open vSwitch to provide the advanced capabilities of container networking [Ove22] [Iva17].

`CLONE_NEWNET` A new network namespace can be created with the syscalls `clone` or `unshare` using the `CLONE_NEWNET` flag.

```c
// Create a new net namespace
if (unshare(CLONE_NEWNET) == -1) {
    return EXIT_FAILURE;
}
```

Figure 2.4.: Example of creating a new network namespace using the *standard c library*

UTS namespace

The name of the Unix Timesharing system (UTS) namespace is misleading largely because its function changed so much over time that it does not reflect its purpose anymore. This namespace provides isolation of two system identifiers: hostname and Network Information System (NIS)[11] domain name [Lin24u].

Inside a child namespace, when using the appropriate syscalls to set the hostname (i.e. syscall `sethostname`) only processes inside that namespace will see the changed hostname.

`CLONE_NEWUTS` A new UTS namespace can be created with the syscalls `clone` or `unshare` using the `CLONE_NEWUTS` flag.

```c
// Create a new uts namespace
if (unshare(CLONE_NEWUTS) == -1) {
    return EXIT_FAILURE;
}
```

Figure 2.5.: Example of creating a new UTS namespace using the *standard c library*

New UTS namespaces are created with the hostname and domain name of the caller's UTS namespace [Lin24u].

User namespace

This namespace isolates security-related properties like user IDs, capabilities and more. User and group IDs can be different inside and outside a namespace, including having a non-privileged user ID outside a namespace and a privileged user ID (i.e., user ID 0) inside the namespace, giving it full privileges within the namespace. User namespaces

[11] The Network Information Service, is a client–server directory service protocol for distributing system configuration data between computers on a unix computer network [Rhe03]

can be nested up to 32 layers. Each namespace can have multiple children namespaces. Each process belongs to exactly one user namespace. [Lin24t]

Capabilities Since Linux 2.2 the privileges associated with the superuser are divided into distinct units called Capabilties [Lin24a]. A process created within a new User namespace is created with full capabilities. The capabilities of a process inside a namespace also apply to all child namespaces of that namespace. However these capabilities are limited to the namespace and its decendents, they do not extend to the initial namespace or other namespaces that are not decendents of the namespace. The capabilities are also limited to the resources owned by the namespace, such as mounted filesystems or managing cgroups. [Lin24t]

Interaction with other namespaces When a process creates new non-user namespaces, the process's user namespace becomes the owner of those namespaces. If the User namespace is created in combination with other namespaces (e.g. `clone` with multiple `CLONE_NEW` prefixed flags), then the User namespace will be created first and receives privileges over the other namespaces. [Lin24t]

ID Mappings User and group ID mappings between namespaces are visible in the `procfs` (2.2.2) files `/uid_map` and `/gid_map` inside the process directory. These files define how IDs in one namespace are mapped to IDs in another. [Lin24t]

File Permissions To determine permissions when an unprivileged process accesses a file, the process credentials (User ID, Group ID) and the file credentials are mapped to what they would be in the initial user namespace and then compared to determine the permissions that the process has on the file. [Lin24t]

`CLONE_NEWUSER` A new mount namespace is created using either `clone` or `unshare` as shown in figure 2.6. When creating a User namespace using the Standard library it is necessary to also write the user and group ID mappings into `procfs`. [Lin24t]

Alternatively it is also possible to use the **command line utility `unshare`**. This utility provides a highlevel interface that includes the mapping of gid and uid.

```c
// Create new user namespace
child_pid = clone(child_func, STACK_SIZE, CLONE_NEWUSER | SIGCHLD, NULL);
if (child_pid == -1) {
    exit(EXIT_FAILURE);
}

// Writing UID and GID mappings to map current user to root inside the namespace
write_mapping("/proc/self/uid_map", 0, getuid(), 1);
write_mapping("/proc/self/gid_map", 0, getgid(), 1);

// `write_mapping` opens a file and writes the mapping in the file
// fprintf(map_file, "%d %d %d\n", inside_id, outside_id, length)
```

Figure 2.6.: Example of creating a new user namespace using the *standard c library*

```
unshare --user --mount --map-root-user bash
```

Figure 2.7.: Example of `unshare` command line

In figure 2.7 the `unshare` command line utility unshares the current user namespace and maps the current user to root inside the new namespace. [Iva17]

- `--user`: Unshares the user namespace.
- `--mount`: Unshares the mount namespace, necessary if you want to remount certain filesystems as if you were root.
- `--map-root-user`: Maps the current user to root (UID 0) inside the new user namespace, allowing for operations typically restricted to the root user within this namespace.
- `bash`: The command to run inside the new namespace, which in this case starts a new Bash shell.

Mount Namespace

This namespace provides isolation of the list of mounts seen by the processes in each namespace instance.

Linux 2.4.19 introduced per-process mount namespaces, and because of that every process has its own view of mounted filesystems in `procfs`. The formerly used `/proc/mounts` now is a symbolic link to `/proc/self/mounts`. The visible mounts in `/proc/<pid>/mounts` is dependent on the processes mount namespace. [Lin24p]

The isolation of the namespace can be loosened, mounts can be created with having so called `propagation types`. The most isolated would be `MS_PRIVATE` and `MS_UNBINDABLE` as they cannot be shared in a peer group (i.e. with other namespaces). The opposite would be `MS_SHARED`, this namespaces is shared within a peer group.

Members are added to a peer group when a mount is marked as `MS_SHARED` and either:

- the mount is replicated during the creation of a new mount namespace; or
- a new bind mount is created from the mount.

In both of these cases, the new mount joins the peer group of which the existing mount is a member. [Lin24j]

`CLONE_NEWNS` A new mount namespace is created using either `clone` or `unshare`:

```c
// Create a new mount namespace
if (unshare(CLONE_NEWNS) == -1) {
    return EXIT_FAILURE;
}
```

Figure 2.8.: Example of creating a new mount namespace using the *standard c library*

IPC Namespace

Interprocess Communication (IPC) is a group of features provided by operating systems to enable data sharing between processes. Most of these features are very lightweight working completely inside the kernel of the operating system or can also be literally just a memory space accessible by multiple processes, which makes them very efficient and highly performant.

The **IPC namespace** isolates IPC resources like System V[12] IPC objects and POSIX message queues. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem. The POSIX message queue interfaces are located in `procfs/sys` and are distinct in each IPC namespace. IPC Objects do not hold the namespace open. When the namespace is freed all IPC objects are automatically destroyed. [Lin24g]

[12] Name of IPC mechanisms that are widely available on UNIX systems: message queues, semaphore, and shared memory [Lin24r]

CLONE_NEWIPC A new IPC namespace is created using either `clone` or `unshare`:

```c
// Create a new IPC namespace
if (unshare(CLONE_NEWIPC) == -1) {
    return EXIT_FAILURE;
}
```

Figure 2.9.: Example of creating a new IPC namespace using the *standard c library*

2.2.5. Control Groups

Control groups, often abbreviated as cgroups, are a Linux kernel feature that limits, accounts for, and isolates the resource usage of a collection of processes. Cgroups allow fine-grained control over various types of resources, including CPU time, memory usage, disk I/O, and network bandwidth. This control is essential in a containerized environment to ensure that each container operates within its allocated resources and does not interfere with other containers or the host system.

cgroups v1 and v2

Cgroups were first released in Linux Kernel 2.6.24 and have evolved over time to include various controllers for resource management. However, the inital version (cgroups v1) has issues with inconsistency and complexity due to uncoordinated development. To fix this, an experimental version of cgroups v2 were added in Linux 3.10 and made official in Linux 4.5. [Lin24b]

While cgroups v2 is designed to replace v1, the latter remains supported for compatibility and v2 has yet to implement all controllers that v1 provides. Because of that it is possible to decide on a per controller basis to use v1 or v2. [Lin24b]

The cgroups used for containerization are all implemented and container runtimes like `crun` (see 2.3.1) fully support operations with cgroups v2. [Wal20]

Controllers

The table 2.1 shows the cgroup controllers supported in both v1 and v2, v1 does support more controllers than listed here.

v2	v1	Description
cpu	cpu	Cgroups can be guaranteed a minimum number of "CPU shares" when a system is busy.
cpu	cpuacct	This provides accounting for CPU usage by groups of processes
cpuset	cpuset	This cgroup can be used to bind the processes in a cgroup to a specified set of CPUs and NUMA nodes.
freezer	freezer	The freezer cgroup can suspend and restore (resume) all processes in a cgroup
hubetbl	hugetbl	This supports limiting the use of huge pages by cgroups.
io	blkio	This cgroup controls and limits access to specified block devices by applying IO control in the form of throttling and upper limits against leaf nodes and intermediate nodes in the storage hierarchy.
memory	memory	The memory controller supports reporting and limiting of process memory, kernel memory, and swap used by cgroups.
perf_event	perf_event	
pids	pids	This controller permits limiting the number of process that may be created in a cgroup (and its descendants).
rdma	rdma	The RDMA controller permits limiting the use of RDMA/IB-specific resources per cgroup.

Table 2.1.: Available Controllers in cgroups v1 and v2 [Lin24b]

2.2.6. Embedded Linux

Embedded Systems describes the use of processors in specialized use cases, from the music player of a toy, traffic lights to controlling complex machinery in factories. All of these contain embedded computers. And these can span from simple 4-bit microcontrollers running small assembly programs from their ROM to general purpose computers running operating systems based on Linux and Windows.

Embedded Linux covers that last part, running on more powerful embedded computers. But while these are more powerful than microcontrollers they’re still very limited compared to the servers powering data centers.

For container runtimes, embedded Linux provides a robust platform for deploying containerized applications on resource-constrained devices. The use of container technology in embedded systems can enhance modularity, improve updateability, and improve security and control by leveraging the isolation mechanisms provided by Linux namespaces, control groups, etc.

2.3. OCI Specifications

The OCI created three specification, based on material donated by Docker in 2015, covering runtime, image format and distribution of containers. These specifications

and the reference implementations are in continuous development and have changed significantly since 2015.

2.3.1. Runtime

The OCI runtime specification defines the configuration, execution environment and lifecycle of a container. At the center of the specification is a set of common container operations and a container configuration that specifies the isolation configuration with support for different platforms and an execution environment. As of specification version 1.2.0 the following platforms are included: Linux, Windows, Solaris, VM, z/OS.

Interface

Out of scope for the OCI runtime specification is the definition of an interface. A basic set of operations are defined, and those operations resemble a Command Line Interface, specifying command names and parameters, but it is explicitly mentioned that these do not serve as the definition of an interface [Ope24c, Runtime and Lifecycle, Operations].

Instead *container managers* have adopted a common interface without a formal standardization. Docker (using Containerd), Podman (using conmon) and similar container managers are compatible with the Command Line Interface (CLI) of runc, the reference implementation of the OCI runtime specification. Alternative implementations of the specification like crun and youki have copied the interface of runc and as such are interoperable with container managers that expect the runc CLI.

Container Configuration

This specification defines a configuration file inside the filesystem bundle: `config.json`. The configuration contains metadata necessary to implement standard operations against the container, like the process to run, environment variables to inject, namespaces to use, etc [Ope24c, Configuration].

Implementations

Currently there exist a few different implementations of the OCI runtime specification. A selection of relevant implementations follows:

runc is the reference implementation of the OCI runtime specification. It was donated by Docker Inc. in 2015, based on Docker 1.6 and the then new libcontainer package that became the basis for runc. This is the default runtime for most high level container software. It is the most mature implementation of the OCI runtime specification and is written in Go. [Ope24e]

crun is a lightweight and fast OCI runtime written in C. It is designed to be highly efficient in terms of both performance and memory usage. `crun` is very suitable for environments where resource efficiency is critical, such as embedded systems and high-density container deployments. [git24]

youki is a relatively new OCI runtime written in Rust. It aims to provide a modern and secure alternative to existing runtimes. ‘youki‘ leverages Rust’s memory safety guarantees to reduce the risk of security vulnerabilities. While still in its early stages, ‘youki‘ is gaining attention for its focus on security and potential performance benefits. It supports the core features of the OCI runtime specification and is designed with a modular architecture to facilitate future extensions. [you24a]

2.3.2. Image

The OCI image specification defines a *container image* (from here on referred to as OCI image), which is the file bundle that enables the high portability of containers. A OCI Image consists of multiple files with metadata concerning the image itself, a set of filesystem layers and a configuration. These are the building blocks to build the *filesystem bundle* (cf. 4.3.2) and the contained root filesystem used for the container by the container runtime. [Ope24a]

The creation of a filesystem bundle from a OCI image is implemented in container managers or special purpose application bundle builders (cf. [Ope23]) and not in OCI

container runtimes, which means that the OCI image specification is of no further concern for this thesis.

2.3.3. Distribution

The *OCI distribution specification* defines an Application Programming Interface (API) protocol to standardize the distribution of OCI images [Ope24b] and other content [Ope24d].

The most prominent content are OCI images (2.3.2), container manager like Docker and Podman uses *registries* that are compliant with the OCI distribution specification to retrieve an OCI image from a remote location.

Because the *distribution specification* is completely handled by either remote software and the *runtime caller* it is of no further concern for this thesis.

3

Related Work

This short chapter creates an overview of a selection of different approaches to deploying containers on embedded systems.

3.1. Northstar

Northstar is an embedded container runtime prototype developed for Linux systems. It focuses on performance and resource efficiency in embedded environments. Northstar containers, known as NPKs, are inspired by the Android APEX technology (cf. [Goo24a]) and contain a root filesystem in SquashFS format along with a manifest file that includes process configuration and meta-information. Northstar Containers are not OCI compliant or compatible and the Northstar runtime **does not support OCI** images.

Northstar has a similar isolation approach to OCI containers, using Linux features like namespaces (mount, PID, IPC, UTS) and cgroups (memory, CPU). The manifest provided with the NPK also contains environment variables, user and group IDs and bind mounts similar to a OCI image manifest or container configuration. Different is that the NPK manifest can also specify capabilities and seccomp configrations which are part of the OCI Container runtimes responsibilities. Northstar explictly states that portabilty is not considered and that Northstar containers are tailored to specific systems.

Comparison to OCI Container Runtimes

- **Portability:** OCI runtimes are designed for portability, allowing containers to be run across diverse environments without modification. In contrast, North-

star's containers are not portable, as they are optimized for specific system configurations, including UID/GID and mount setups.

- **Resource Management:** OCI runtimes offer comprehensive resource management capabilities, often integrating with container orchestrators like Kubernetes to handle resource allocation and scaling. Northstar, however, is geared towards embedded environments where resources are more constrained and configurations are less dynamic. This becomes apparent as the resource restriction are specified in the NPK Manifest, while OCI Container have resource restriction set in the runtime.

- **Use Case Focus:** OCI runtimes are general-purpose and applicable across various industries and environments. Northstar is specifically designed for embedded systems, where efficiency and performance are paramount, and the runtime can be closely tailored to the hardware and software stack in use.

Overall, while OCI Container Runtimes offer broad applicability and adherence to standards, Northstar provides a specialized solution tailored to the unique requirements of embedded systems. [Acc24]

3.2. VxWorks

VxWorks does not have a lot of technical details public as it is a closed source and expensive RTOS[1] that is geared toward critical infrastructure like military technology, airplanes, robotics and spacecraft. The company behind VxWorks, Wind River Systems, states on their website that VxWorks is the first and only RTOS that supports application development with containers and that it **does support the OCI container standard**. [Win24]

3.3. Pantavisor

Pantavisor is a framework for containerized embedded Linux by Pantacor (cf. [Pana]). Pantacor claims that Pantavisor is the easiest way to build and manage embedded Linux projects with lightweight containers [Pan21].

[1] Real Time Operating System

Pantavisor is designed for embedded Linux with a tiny memory footprint, and has no performance hit, claiming that the containerized applications run equal to their bare metal counterparts. Panatavisor is based on LXC and mentions support for *Docker* containers. [Panb]

4

Design

This chapter will concern itself with the analysis and specification of requirements for the runtime that will be implemented in Chapter 5. It will do so by first establishing an overview of the runtime and its environment. Then analyse the goals defined in the introduction (1.2) and preexisting limits. Based on that a set of requirements will be created and finally this chapter will go though relevant sections of the OCI runtime specification defining the described behavior for this implementation.

4.1. Overview

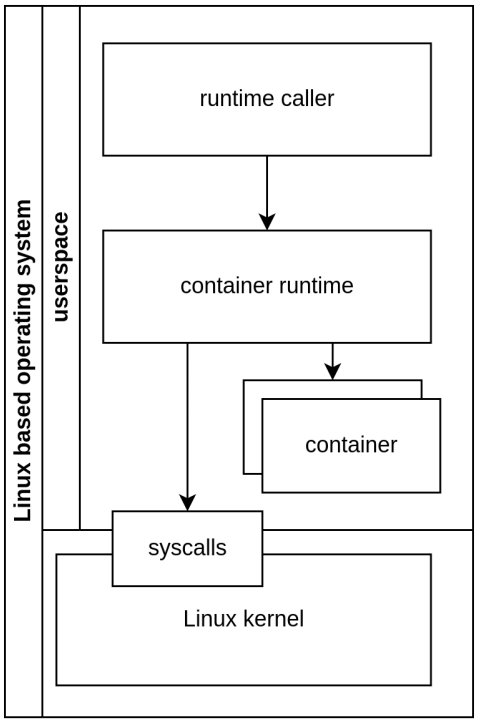

Figure 4.1.: Sketch of high-level components

The design of the runtime will only consider the Linux container as defined by [Ope24c, Configuration] and [Ope24c, Linux Container Configuration].

The figure 4.1 shows a high-level overview of the environment for the runtime. The container runtime is implemented in Chapter 5. The runtime caller is defined in Section 4.3.1. The syscalls used by the runtime are explained in the theoretical background in Section 2.2.1, and implementation details are described in Chapter 5.

4.2. Specification

Designing a runtime that will implement a subset of the OCI runtime specification, but at the same time should still be interoperable (in the following sections also described as *compatible*) with container managers requires some compromises. This section will define requirements based on the goals defined in this thesis and the limits set by the scope of this bachelor thesis.

Two requirements can be taken directly from the goals defined in the introduction (1.2):

1. The container runtime must support Linux, and that will also be the only platform supported.

2. The container runtime must have a minimal viable compatibility with a container manager.

The *minimal viable compatibility* mentioned in the second requirement means that the implementation will be a drop-in replacement for the default runtime of the container manager for the base operations defined in the OCI runtime specification.

One of the primary goals of this implementation is to have a minimal containerization that provides similar portability, compatibility and automation like a standard OCI container. Which means that the operations and parameters expected by the *runtime caller* (cf. 4.3.1) cannot be modified. Also the runtime will have to work with the filesystem bundle (cf. 4.3.2) as provided by the runtime caller.

This implementation assumes that software on embedded systems can be considered trusted software running on trusted hardware with trusted data. Based on this assumption, this implementation will sacrifice a lot of the security features of standard OCI containers. This will make the runtime very susceptible to attacks like container escapes (cf. [Ant19]).

Also the container networking that is a big part of systems like Docker and a important dependency for systems like Kubernetes will not be implemented, the containers will be directly exposed to the host networking stack, sharing it with processes on the host and other containers. This will have a negative impact on the portability of the container because the processes inside the container that interact with the network stack will have to take the host into account when building the container.

4.3. OCI Runtime Specification

This section discusses the requirements, compromises and implementation specific decisions regarding which parts of the OCI runtime specification will be implemented in Chapter 5. This section is loosely structured after the *Configuration* and *Linux Container Configuration* sections of the OCI runtime specification [Ope24c].

4.3.1. Runtime Caller Compatibilty

The OCI runtime specification defines *runtime caller* as:

> An external program to execute a runtime, directly or indirectly
>
> — *[Ope24c, Glossary]*

The implementation will during development and testing only take a single *runtime caller* into consideration. For this purpose the container manager **Podman** was selected.

Podman (cf. 2.1.4) has two important differences to Docker. While technically also being a *indirect runtime caller* like Docker, it is in fact a lot closer to being a *direct runtime caller*, because it only uses the very lightweight utility conmon that also shares a similar CLI based interface to runc (cf. 2.3.1) while docker has multiple layers of abstraction between user and runtime. And the second important difference is its own overhead and feature set, Podman is a lot more suited for standalone autonomous deployments in comparison to Docker.

Podman is by default compatible with container runtimes that are compliant with the OCI runtime specification and provide a *runc-like* CLI (cf. 2.3.1).

This means that in theory the implementation in Chapter 5 should be compatible with other *runtime callers* that expect a *runc-like* CLI, but the verification of this is not in scope for this thesis.

4.3.2. Filesystem Bundle

The filesystem bundle is a set of files containing all the necessary data for a compliant runtime to perform all standard operations against it. It should contain at the top level a `config.json` file with configuration data and the container's root filesystem. [Ope24c]

The root filesystem will contain all dependencies that were packaged in the OCI image, but some system resource like the virtual filesystem `procfs` have to be mounted or created during the container initialization. [Ope24c]

Podman, other container managers and specialized bundle builders implement the creation of the filesystem bundle based on the OCI image specification [Ope24a] and OCI runtime specification, but the implementation of the filesystem bundle creation is not part of the OCI runtime specification. The `CREATE`-Operation (5.5.1) of the container runtime is provided with a path to the existing filesystem bundle. [Ope24c]

4.3.3. Errors & Warnings

The OCI runtime specification does not state how or if an error or warning is exposed to the *runtime caller*, but it states that

> generating an error MUST leave the state of the environment as if the operation were never attempted - modulo any possible trivial ancillary changes such as logging
>
> - [Ope24c, Runtime and Lifecycle, Errors]

and for warnings that

> Unless otherwise stated, logging a warning does not change the flow of the operation; it MUST continue as if the warning had not been logged.
>
> - [Ope24c, Runtime and Lifecycle, Warnings]

Because this implementation will explicitly not implement certain features and requirements, it would need to throw errors in a lot of places, making the runtime unusable.

Instead the implementation will throw warnings in place of errors when the breaking of the specification is intentional.

For the implemented behavior see 5.2.

4.3.4. Operations

This implementation will contain all operations specified in the OCI runtime specification: `CREATE`, `START`, `KILL`, `DELETE` & `STATE`.

As a result of the runc-like interface and the requirement to be compatible with Podman, a number of additional operations and options will have to be implemented in this runtime implementation. Only the necessary of these additional operations and options will be implemented. For example conmon, the tool used by Podman to interact with the OCI runtime, expects the flag `--pid-file value` to specify exactly where the PID of the container/init process is returned.

The spec defines a set of operations that must exist to be compliant. But container managers expect more than what is defined in spec, for example, conmon expects the init process PID to be written to a specific file.

4.3.5. Mounts

The runtime will attempt all specified mount operation in the container configuration. It will be treated as a warning instead of an error if a mount operation fails and the container creation will not be aborted.

4.3.6. Out of scope

The next sections will describe parts of the OCI runtime specification that will be declared partly or completely out of scope for the implementation in Chapter 5.

Default Filesystems

In [Ope24c, Linux Container Configuration, Default Filesystems] are multiple filesystems specified that should be made available in each container's root filesystem. This will not be implemented, the implementation will assume that all necessary filesystems are specified as mounts in the container configuration. This is not compliant with the OCI runtime specification.

Namespaces

The PID, UTS and user namespace should be implemented as specified. The namespaces IPC, network (cf. 4.3.6), cgroup and time will be created but any additional steps that may be required to finalize the namespace will not be implemented.

Network The containerization of networking is one of the most powerful and most complex features of a container and very important for systems like Kubernetes and PaaS provider. But it depends on additional configuration and setup, using switch software like Open-vSwitch and a lot of additional setup to configure the networking inside the container, outside the container and the required bridging between networks.

So because of it's complexity and the reduced relevance in embedded systems it **will not be implemented**. Instead processes will use the host networking stack, like running the process outside of a container.

Devices

In [Ope24c, Linux Container Configuration, Devices] are a list of devices in the container configuration and a list of default devices defined, which must be made available in the container. This will not be implemented.

Control Groups

The cgroup isolation defined in [Ope24c, Linux Container Configuration, Control Groups] will not be implemented.

Other Configuration

The OCI runtime specification has more sections describing the container configuration which pertain to things that are out of scope for this thesis.

Also these section of [Ope24c, Linux Container Confifuration] will not be implemented:

- Default Filesystems
- Offset for Time Namespace
- Unified
- IntelRdt
- Sysctl
- Seccomp
- Rootfs Mount Propagation
- Masked Path
- Readonly Paths
- Mount Label
- Personality

5

Implementation

This chapter describes the implementation of the container runtime and provides additional context to the source code. The first section will provide a high level overview of the software architecture, the chosen programming language, used third party libraries and a small overview over the directory structure. The following section will explain details and decisions of the implementation in regards to logging, the API, state management, specification specific details, ipc, the implementation of the isolation and the container lifecycle.

The implementation of the reduced feature set of the OCI Runtime specification will be called ***R**educed **OCI** Runtime* (roci). And the OCI version specified in places like the state output (cf. 5.3.1) is suffixed with the tag `+modified`, resulting in the OCI version: *1.2.0+modified*.

5.1. Architecture

The sketch in figure 5.1 shows a high-level overview of the planned architecture for the runtime, consisting of the runtime executable and two directories important for the container lifecycle. The runtime binary will implement two processes, the *cmd process* and *init process*.

The **cmd process** is providing the CLI for the runtime caller and is responsible for the state management of containers.

The **init process** is almost completely responsible to construct the container based on the provided container configuration (`config.json`). One exception being the pid namespace that is not unshared by the init process but created during the cloning of the process in the cmd process.

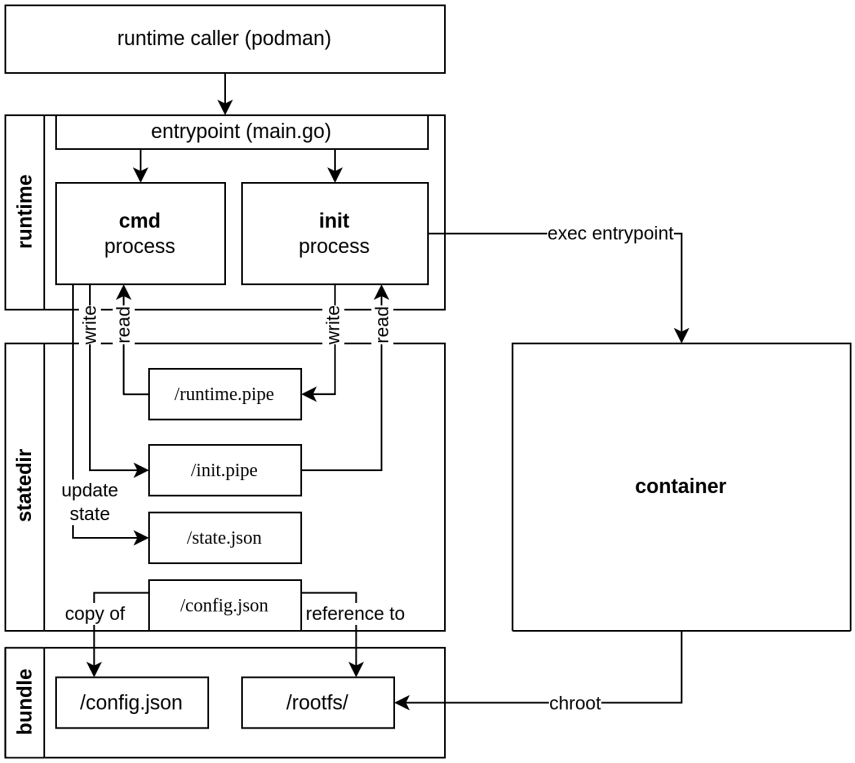

Figure 5.1.: Runtime architecture diagram

The **container state directory (statedir)** is managed by the cmd process and contains the named pipes (cf. 2.2.1) used for IPC between the *cmd* and *init process*, a copy of the container configuration (`config.json`) and a json document used for state management (`state.json`).

While not directly part of the runtime, the **filesystem bundle** provided by the runtime caller is used by the container runtime for the source of the container configuration and the root filesystem of the container. The arrow going from *container* to */rootfs/* in figure 5.1 does not mean that the container process is executing chroot but that the root of the container is rootfs.

5.1.1. Language

The programming language Go (also called Golang) has emerged as the most suitable language for the implementation of roci. Go is widely used in the container landscape with the reference implementation of the OCI runtime specification being written in go and also software like Kubernetes that is based on containers being written in go. Go has robust community support with a lot of libraries, and is very suitable for fast

development and easy integration because of its great cross platform compatibility. [Goo19]

However it is important to note that a low-level language like C or Rust would be more efficient and a production ready runtime for embedded Linux systems should take this into consideration. Also, because a lot of the actual isolation is based on Linux features like namespaces, which require syscalls as an API, the go implementation depends on a module written in C that performs these syscalls [you24b].

Despite these considerations, for the purpose of this thesis, the advantages of Go make it the preferred choice.

Go specific features

Go provides some language features that are uncommon and their functionality may not be instantly understood. The use of these features in the implementation is minimal and occurrences will refer to specific gobyexample.com pages for further explanation of these features: [MBa].

5.1.2. Third Party Libraries

Third party libraries are specified in the go.mod file at the root of the project. The *require* statements are separated into direct and indirect dependencies. The direct dependencies are the ones directly used in the source code of the runtime and the next sections will provide more details about these dependencies. The indirect dependencies are used by the direct dependencies, these will not be further explored in this thesis.

Go dependencies are called packages and are identified by a unique url. Most of the times the url will lead to a git project with the source code of the package.

github.com/opencontainers/runtime-spec

The OCI runtime specification github project also provides data structure definitions for Go and the project itself is setup to be imported as a package into projects. This

will be used to guarantee specification compliant serialization and deserialization of JSON documents, which is important for interoperability with *runtime callers*.

github.com/spf13/cobra

Because roci is going to implement a runc-like interface, it will need a CLI. To provide a stable and proven CLI experience the library *Cobra* will be used. Cobra is a go package for creating CLI applications and is widely used in the Cloud Native Landscape[1], being used in projects like Kubernetes and Github CLI and provides are declarative struct based framework to build a CLI. [cob24a]

It comes together with its own command line tool, called *Cobra Generator* [cob24b]. This tool creates a base project structure and helps with adding additional commands.

The project structure is created with `cobra-cli init`. Using this command will modify the `main.go` file, the entrypoint of the go program, to add code that executes the *root command* inside the newly created `cmd/` package. Inside the `cmd` package the new file `root.go` declares the *root command*. The root command is the root node of the CLI command tree. All subcommands will be added to the root command.

```
1 // rootCmd represents the base command when called without any subcommands
2 var rootCmd = &cobra.Command{
3    Use:   "roci",
4    Short: "Reduced Open Container Initiative (roci) runtime",
5    Long: `roci is a experimental runtime only intended for testing...`,
6    Run: func(cmd *cobra.Command, args []string) { },
7 }
```

Figure 5.2.: Root command of a cobra command tree

Figure 5.2 shows the root command generated by cobra. The struct fields `Use`, `Short` and `Long` add documentation to the *help command* output (i.e. `--help`, `roci help` or on error).

The `Run` field assigns the function that implements the command. It is executed with the Arguments passed to the program via command line, but already processed according to cobras rules. These rules include that the command names are stripped, the flags are parsed and removed from the arguments array, and more also depending on developer configuration.

[1] The term *cloud native landscape* describes the wide variety of software used in and around cloud technology [Clo24]

For example if a subcommand is added with the name `test` to the *root command*, and the program is called like `<executable name> test --flag flagvalue ARG1 ARG2` then the arguments passed to the `Run` function of the test subcommand struct would be `[]string{"ARG1", "ARG2"}` and the parsed flag would be accessible via the cobra.Command struct passed to the `Run` function.

Flags are defined by adding them to a flag set inside the init function of the command file.

```
1 func init() {
2    createCmd.Flags().StringP("bundle", "b", ".", `path to the root ...`)
3    createCmd.Flags().String("pid-file", "", `specify the file ...`)
4 }
```

Figure 5.3.: Declaration of command flags

In go the `init()` function is a static function that is executed before the entrypoint of the program is executed. Cobra uses these function to add additional non-constant (i.e. code that can not be defined in a `const` field) configuration to the command. To add a flag to the command a flagset is used. This object allows the difintion of flags with static type definitions in two variants. Using a parameter with a string type, it can be defined with `.String` or `.StringP`, the former provides the possibility of adding a short flag name. Besides that the parameters for the flag defintion are the same: name of flag, default value, and description for help output. To use the flag inside the `Run` function of the command it must be retrieved from the flag set:

```
1 value, err := cmd.Flags().GetString("flag-name")
```

Figure 5.4.: Get value of flag

Cobra provides a lot more fields that enable advanced configuration like `PreRun` (running before `Run`), `PostRun` (running after `Run`), `Args` (doing arguments validation like number of arguments), and more.

github.com/spf13/viper

Viper is part of the cobra toolkit and is a configuration solution for apps. It provides a very simple interface to retrieve config values from multiple sources with appropriate override-rules (e.g. command line flag overrides value in file). [vip24]

With an easy setup during initialization that can also be generated by the cobra generator tool (cf. 5.1.2).

```
func initConfig() {
    // specify config file
    viper.AddConfigPath("/etc/roci")
    viper.SetConfigType("yaml")
    viper.SetConfigName("config")

    // default values
   viper.SetDefault("containerDir", "/run/roci/container")

    // read in environment variables that match
   viper.AutomaticEnv()

   // If a config file is found, read it in.
   if err := viper.ReadInConfig(); err == nil {
      fmt.Fprintln(os.Stderr, "Using config file:", viper.ConfigFileUsed())
   }
}
```

Figure 5.5.: Viper init

After the initialization the config values can be retrieved via simple helper functions like `viper.GetString("containerDir")`. The returned value will be the config value that *won*.

For example if a value is defined in the config file and also as in a environment variable, then the value in the environment variable wins according to the rules defined by viper. And the helper function would then return the value of the environment variable.

go.uber.org/zap

Zap is a high performance structured logging library for go. It provides a simple interface and a lot of configuration options. [Ube24]

Protocol buffers

Protocol buffer (protobuf), is a language-neutral, platform-independent mechanism for serializing structured data. It functions as both an Interface Definition Language (IDL) and a method for efficiently encoding data, offering a more compact and faster alternative to XML and JSON. The process involves defining data structures in .proto files, including data types, field numbers, and rules for required or optional fields.

These .proto files are then compiled into code in various programming languages, enabling seamless data exchange between systems. Protocol Buffers are very suitable for serial communication because of their efficiency in serialization, deserialization, and overall data size. [Goo24b]

```
1 syntax="proto3";
2
3 package proto.init.v1;
4 option go_package="proto/";
5
6 message Example {
7  int32 field_name = 1;
8 }
```

Figure 5.6.: Example .proto file

In 5.6 a message with the type name *Example* is defined with a single field with the name *field_name* and of an integer type with a 32 bit size.

A protobuf compiler like protoc is then used to generate the code for the desired programming language, in this case Go. It then generates the language appropriate data structure using standard code styles. For example the message defined in figure 5.6 resulted in the go code in figure 5.7. [Goo24b]

```
1 type Example struct {
2   state     protoimpl.MessageState
3   sizeCache   protoimpl.SizeCache
4   unknownFields protoimpl.UnknownFields
5
6   FieldName uint32 `protobuf:"varint,2,opt,name=fieldName,proto3" json:"fieldName,
        omitempty"`
7 }
```

Figure 5.7.: generated protobuf code

The fields `state`, `sizeCache` and `unknownFields` are private and used by the serialization and deserialization tooling for protobuf. [Goo24b]

In this case the **go library google.golang.org/protobuf** is used. This is the official implementation by google, the developers of protobuf. This library provides a simple way to transform the protobuf messages between *Go* structs and binary format. [Goo24c]

5.1.3. Project Structure

Some programming languages or frameworks come with a predefined directory structure for the source code but *Go* does not do that. This section will establish a common understanding of the source code structure and the purpose of said structure. Each paragraph identified by the name will explain a file or directory in the project directory. Subdirectories containing *Go* that can be important into go source files are called packages.

main.go In this file is the `main` function defined that implements the entrypoint of the executable.

go.mod This file contains metadata like the project name, *Go* version and the required dependencies (cf. 5.1.2)

cmd This package is generated by Cobra (cf. 5.1.2) and contains the implementations of the CLI commands.

pkg This is the top level directory of all the non-cmd packages defined in this project.

pkg/libcontainer The package name is inspired by the `libcontainer` package of runc, and while having a similar approach and use, runc's libcontainer and roci's libcontainer are very different in their implementation. This package contains the core of the containerization effort, implementing the namespace isolation, interprocess communication and state handling.

5.2. Logging

The OCI runtime specification defines two cases for unexpected behaviors: errors and warnings (cf. 4.3.3). And while one important distinction is made between the two (on

warning the execution continues, on error the execution is aborted), there is no further defined behavior in the specification.

This implementation defines these exceptions as specified in Table 5.1. The table defines in addition to errors and warnings the behavior of debug log output.

Exception	**Log Level**	**Behavior**
Error	error	logs error and returns with a non-zero exit code
Warning	warn	logs warning and continues execution
	debug	logs to console when verbosity is enabled and continues execution

Table 5.1.: Defined behavior of Errors and Warnings and debug output

The log level refers to the log level of the zap logging library (cf. 5.1.2). Error and warning logs will be printed to `stdout`, log levels below that will only be printed if verbose configuration is enabled. And during the benchmark in Section 6.2 logging is completely disabled.

When the execution is stopped in case of an *error* exception an error will be returned to the user. The returned error is either a generic error generated by go or a specific error defined in `model/errors.go`. The list of defined errors in figure 5.8 is copied from the reference implementation runc (cf. [Ope24e]). This is done to ensure interoperability with runtime callers that also expect *runc-like* errors.

```
var (
   ErrExist      = errors.New("container with given ID already exists")
   ErrInvalidID  = errors.New("invalid container ID format")
   ErrNotExist   = errors.New("container does not exist")
   ErrPaused     = errors.New("container paused")
   ErrRunning    = errors.New("container still running")
   ErrNotRunning = errors.New("container not running")
   ErrNotPaused  = errors.New("container not paused")
)
```

Figure 5.8.: Defined errors

5.3. API

The OCI runtime specification states that it does not specify any API, but that *runtime callers* often execute a runtime via a CLI that consists of the same commands, arguments and flags as the runc CLI [Ope24c, Glossary, Runtime Caller].

This implementation will adopt a runc-like (cf. 2.3.1) interface, implementaing the minimal required commands, arguments and flags of runc to be interoperable with Podman and other container managers that depend on a runc-like interface.

5.3.1. CLI

The Command Line Interface is split into two, these two parts are the two processes that make up the runtime. The **cmd process** implements the cli used by the runtime caller and serves the container operations as defined in 5.5. The ***init process*** handles the container initialization, basically building the container around itself. The separation between the two processes is made instantly after runtime start using the function defined in 5.9.

```
1 func isInitProcess() bool {
2    return len(os.Args) == 3 && os.Args[1] == cmd.InitCommandName
3 }
```

Figure 5.9.: Function that determines init or cmd process

If 5.9 returns true the function `ExecuteInit` in `cmd/init.go` is executed, which is the entrypoint for the *init process* (cf. 5.3.1). If false is returned `ExecuteCLI` in `cmd/root.go` is executed, this is the root command of Cobra (cf. 5.1.2) and the entrypoint of the *cmd process* (cf. 5.3.1).

cmd process

The *cmd process* implements the interface used by runtime callers. As specified in 4.3.1, the designated container manager for this implementation will be Podman (cf. 2.1.4), and because of that a runc-like (cf. 2.3.1) interface will be implemented.

The CLI of the *cmd process* is implemented using the go library cobra (cf. 5.1.2). Cobra provides a struct-based approach to building a Command Line Interface. Each command will have a struct (e.g. Figure 5.2) defining its implementation, and all required parameters. The primary focus of this CLI are the required operations defined in the OCI runtime specification [Ope24c]. All commands besides these required operations are considered unstable and are only to be used for testing and development of the runtime.

In addition runc itself provides a lot more commands and parameters then what is defined in the OCI runtime specification. Including the required operations runc has 17 defined commands, most of them will not be implemented, but some of these are required by runtime callers like Podman and will be implemented as needed. Flags like the global flag `--systemd-cgroup` exist but are marked as *deprecated* using `rootCmd.PersistentFlags().MarkDeprecated("systemd-cgroup", "")` . This is important because if flags like this are undefined they can cause parsing problems when the runtime caller uses them. With the deprecation mark they will be considered during parsing but ignored afterwards.

The CLI consists of the following commands, each paragraph title being the command-line structure of the command and a reference to the attached runc help output:

create [flags] <container-id> (A.1.2)
The create command has the required *positional argument*[2] *container-id* and 2 flags:

- `--bundle, -b` is the second required parameter for this operation defined in the OCI runtime specification but is opional in runc because it uses the current directory as the default value.

- `--pid-file` specifies a special file only used for writing the pid of the init and subsequently the container process into. This flag is part of the runc interface and is expected by Podman.

start <container-id> (A.1.3)
The start command has the required positional argument *containerId*. The containerId must be of an existing container. Besides that neither the OCI runtime specification [Ope24c] nor the runc help specifies (see A.1.3) any additional parameter for this command.

kill <containerid> [signal] (A.1.4)
The kill command has the required positional argument *containerId*. The containerId must be of an existing container.

[2] *Positional arguments* in Cobra (cf. 5.1.2) refers to parameters in a CLI that are passed via the command line but are not prefixed with dashes (i.e. flags)

The second required parameter by the operation defined in [Ope24c] is the signal parameter. This parameter is implemented in runc has a optional second positional argument with the default signal `SIGTERM`. This argument supports signal names and numeric identifiers of signals [Ope24e, `kill.go`].

The runc help (see A.1.4) specifies an additional flag, but it is marked as obsolete in the source code [Ope24e, `kill.go`], is not required by Podman and will therefor not be implemented.

delete [flags] <containerid> (A.1.5)
The delete command has the required positional argument *containerId*. The containerid must be of an existing container that is either in state `CREATED` or `RUNNING`.

The runc help specifies in A.1.5 an additional flag:

- If specified, `--force`, or `-f` send a kill signal to the container before deleting it. Ensuring that the delete operation does not throw an error if the container is still running. This flag is not part of the OCI specification [Ope24c].

state <containerid> (A.1.6)

The state command has the required positional argument *containerId*. The containerId must of an existing container.

If successful, the runtime will return the current state of the container, printing the contents of the `state.json` in the container statedir. This matches the container state schema defined in [Ope24c].

```
1  {
2    "ociVersion": "0.2.0",
3    "id": "oci-container1",
4    "status": "running",
5    "pid": 1312,
6    "bundle": "/containers/redis",
7    "annotations": {
8      "myKey": "myValue"
9    }
10 }
```

Figure 5.10.: Example of container state json

`ContainerPreRunE` This is not a command, but a function defined in `cmd/common.go` used by all of the operations as a `PreRun`-function before executing `Run`-function. The figure 5.11 shows the three steps executed by the `PreRun`-function.

It assures that the calling user has super user privileges, then it creates the required directories on the host and finally initializes the abstraction of the confs (cf. 5.4).

```
if !util.HasSudo() {
    return model.ErrNoSudo
}

var allDirs = []string{viper.GetString(containerDirFlag), viper.GetString(configDirFlag)}
for _, dir := range allDirs {
    err = os.MkdirAll(dir, 0o755)
    if err != nil {
        return err
    }
}

if confs, err = libcontainer.NewContainerFS(viper.GetString(containerDirFlag)); err != nil {
    return err
}
```

Figure 5.11.: cmd process prerun

init process

The init process implements the process containerization (cf. 5.7) and builds the container based on the container configuration (cf. 2.3.1). The interface for the cli is really simple and only has two arguments `init <state-dir>`. The init command is used by the runtime to determine that the init process was requested and the statedir argument is the absolute path to the statedir (5.4.1).

5.3.2. Runtime Configuration

The configurable values are setup using viper (cf. 5.1.2), each configuration value has a configuration key that is then used to retrieve the value from viper. The defined configuration values for roci are listed in table 5.2.

Key	Default Value	Description
containerDir	`/run/roci/container`	Path to the *container filesystem (confs)* (cf. 5.4)

Table 5.2.: runtime configuration

5.4. Container Filesystem

The container filesystem (confs) is the core component of roci. It is implementing all the container operations supported by roci, covering the standard operations defined in the OCI runtime specification and additional operations added by roci itself. It is using the container directory specified in the runtime configuration (cf. 5.3.2) to store the persistent state of containers managed by roci with each entry in the container directory being a statedir (cf. 5.4.1).

All supported container operations are defined in the `ContainerFS` interface (see 5.12) that is implemented in the `libcontainer` package.

```
type ContainerFS interface {
   /* Standard container operations */
   Create(id, bundle string, spec specs.Spec) (container *Container, err error)
   Start(id string) (err error)
   Kill(id string, signal syscall.Signal) (err error)
   Remove(id string) (err error)
   State(id string) (state specs.State, err error)

   /* Additional container operations */
   List() (containers []specs.State, err error)
}
```

Figure 5.12.: ContainerFS interface definition

The implementation of these operations is described in Section 5.5.

5.4.1. statedir

The container state directory (statedir) is the directory in which the container runtime stores its container specific metadata and file descriptors needed for IPC. It can contain up to 4 files (see Table 5.3).

Filename	Category	Description
config.json	configuration	Copy of the configuration file in the filesystem bundle
state.json	state	Stores state information about the init and container process
runtime.pipe	IPC	named pipe, used to send messages to the cmd process
init.pipe	IPC	named pipe, used to send messages to the init process

Table 5.3.: Files inside container statedir

The state.json is based on the schema defined by the OCI runtime specification, see [Ope24c, Runtime and Lifecycle, State].

The IPC files, how they are created and used is described in 5.6.

5.4.2. State manager

The *state manager* type is responsible for loading and interacting with the `state.json` in the statedir. The state manager is either created by `func NewStateManager(stateDir string, state *specs.State)(*StateManager, error)` that writes a new state to the statedir or `LoadStateManager(stateDir string)(*StateManager, error)` that loads an existing state from the statedir.

The state is updated by modifying the state struct stored in the state manager to reflect the desired state and then calling the method `func (s *StateManager)UpdateState()(err error)`, which then overrides the existing state on file with the new state.

5.5. Container Operations

This section describes the operations defined in confs (cf. 5.4) based on the container lifecycle defined in the OCI runtime specification [Ope24c].

In the context of the runtime a container is two things, a state from which a container can be constructed and a process that is either waiting to start the entrypoint or is running the entrypoint. The create operation (5.5.1) is the inception point of both: The container state with the creation of the container state directory (statedir) and the container process with the execution of the *init process*.

The **start operation** (5.5.2) then replaces the *init process* with the *user-specified process* (the entrypoint) and the **kill operation** (5.5.3) is stopping the container process.

After the kill operation or if the container stops on its own the container process stops existing, but the container does not. The container is still represented by the statedir from which the container can be started again.

The **delete operation** (5.5.4) removes the statedir and all container process dependencies which is the actual end of the container lifecycle.

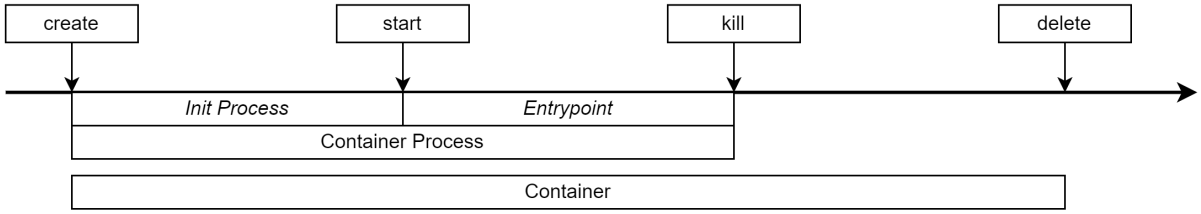

Figure 5.13.: Lifecycle of a container

The following section will describe the operations defined in confs (cf. 5.4) based on the container lifecycle defined in the OCI runtime specification [Ope24c]. Each section will quote parts of the lifecycle and then describe how they are implemented in the confs operation. The confs type is initialized before each operation is executed in the `cmd` package (see 5.11).

5.5.1. CREATE

> 1. OCI compliant runtime's create command is invoked with a reference to the location of the bundle and a unique identifier.
>
> 2. The container's runtime environment **MUST be created according to the configuration in config.json**. If the runtime is unable to create the environment specified in the config.json, it MUST generate an error. While the resources requested in the config.json MUST be created, the userspecified program (from process) MUST NOT be run at this time. **Any updates to config.json after this step MUST NOT affect the container.**
>
> \- [Ope24c, Runtime and Lifecycle, Lifecycle]

The invocation of the create operation is the first step of the lifecycle, as defined in step 1. In roci the create operation is the only operation where the `Create` method of the confs type is not called directly but rather the function in figure 5.14. This is done to separate the different concerns related to the container creation. The confs type is only responsible for the state management in the container directory (cf. 5.4) but the container creation involves steps outside of that concern.

The `CreateContainer` function shown in 5.14 implements the complex container creation process. This process involves four steps:

```
1  func CreateContainer(fs *FS, id, bundle string) (c *Container, err error) {
2     var spec specs.Spec
3     if err = util.ReadJsonFile(path.Join(bundle, model.OciSpecFileName), &spec); err ↩
          != nil {
4        return nil, err
5     }
6     PrepareSpec(&spec, bundle)
7
8     c, err = fs.Create(id, bundle, spec)
9     if err != nil {
10       return nil, err
11    }
12
13    pid, err := c.Init()
14    if err != nil {
15       return nil, err
16    }
17
18    c.state.SetPid(pid)
19    c.state.SetStatus(specs.StateCreated)
20    if err = c.state.UpdateState(); err != nil {
21       return nil, err
22    }
23
24    return c, nil
25 }
```

Figure 5.14.: CreateContainer function

I Prepare container configuration

II Create container *statedir* (cf. 5.4.1)

III Start *init process* (cf. 5.3.1)

IV Conclude create operation

In step I the container configuration is loaded from the filesystem bundle and parsed. Then some modification are made to prepare the configuration for further processing by roci.

In step II the `create` method of the confs is called (see 5.12) and the containerId, path to the filesystem bundle and the parsed OCI container configuration is passed as parameters. To ensure that any changes made to the config.json after the second step of the container lifecycle will not affect the container, the parsed container configuration will be stored on disk inside the statedir. The `create` method returns a `Container` struct (see 5.15).

For step III the `Init` method of the container struct is called. This method will start the *init process* and wait for the ready signal (cf. 5.6), which is send when the container creation is done (see 5.16).

The creation of the container spans step II to III and is described in further detail in Section 5.7.

After the container was created, the PID of the *init process* and the new container status *created* is written into the statedir (cf. 5.4.2), concluding step IV of the `CreateContainer` function and step 2 of the container lifecycle.

```
type Container struct {
   id     string
   state  *StateManager
   config specs.Spec
   initp  *initp.Process
}

func (c *Container) Init() (pid int, err error) {
   return c.initp.Start()
}
```

Figure 5.15.: Container struct with methods

5.5.2. START

> 6. Runtime's start command is invoked with the unique identifier of the container.
>
> 8. The runtime MUST run the user-specified program, as specified by process.
>
> - [Ope24c, Runtime and Lifecycle, Lifecycle]

After the start operation is invoked by step 6 of the lifecycle the *start message* is send to the *init process* (see 5.16) using the IPC defined in 5.6. This tells the *init process* to run the user-specified program of the container which concludes step 7 of the lifecycle.

5.5.3. KILL, crash, exit, ...

10. The container process exits. This MAY happen due to erroring out, exiting, crashing or the runtime's kill operation being invoked.

If the container was stopped by the `KILL` operation, the runtime will change the state instantly to reflect that. But if the container stopped for whatever reason on its own or by other external factors then roci will detect this only when the state operation (cf. 5.5.5) is invoked.

This concludes the end of the container process (see 5.16), but not the end of the container (see 5.13).

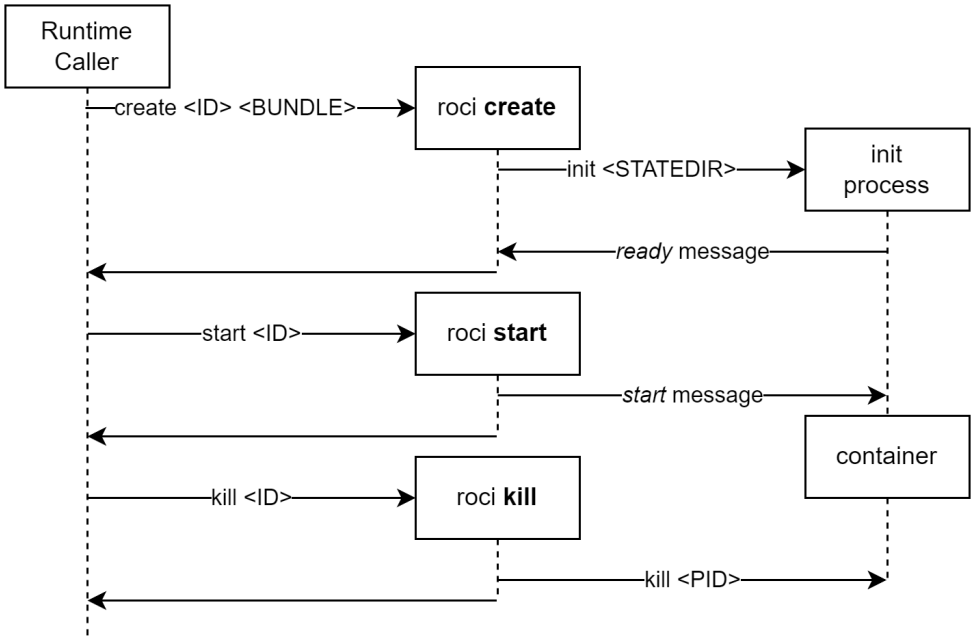

Figure 5.16.: Lifecycle of the container process

5.5.4. DELETE

11. Runtime's delete command is invoked with the unique identifier of the container.

12. The container MUST be destroyed by undoing the steps performed during create phase (step 2).

- [Ope24c, Runtime and Lifecycle, Lifecycle]

After the invocation of the delete operation in step 11 of the lifecycle the rootfs supplied by the runtime caller as part of the filesystem bundle during creation of the container is cleaned of the modifications that were done during the creation process. After that was successfully done the statedir (cf. 5.4.1) is removed.

The container is considered deleted after the statedir was removed (see 5.13).

5.5.5. STATE

The state operation itself is not part of the lifecycle. Its only task is to return the state of the specified container. The process to retrieve the current state of a container has the side effect of also verifying that the container process is still running, if that is not the case the state will be updated accordingly before returning the state of the container to the runtime caller.

OCI Version

The state schema defined by the OCI runtime specification in [Ope24c, Runtime and Lifecycle, State] contains a field with the OCI specification version. Because this runtime is not compliant with the OCI specification a modified version will be returned.

In the OCI runtime specification github project, work in progress versions of the OCI runtime specification use the `+dev` suffix to indicate that it is not a stable version. Based on this, roci will return the version: `1.2.0+modified`.

5.5.6. Hooks

Steps 3, 4, 5, 7, 9, and 13 of the container lifecycle defined in [Ope24c, Runtime and Lifecycle, Lifecycle] specify which set of hooks are to be invoked. These are executed at the appropriate places in the implementation.

5.6. Runtime IPC

Because the runtime consists of two separate processes it is not possible to simply call a function or set a value from the *cmd process* in the *init process* or vice versa. To enable communication between the processes a IPC solution is required.

Table 5.4 is a list of all the messages send between the *cmd process* and the *init process*.

Name	Caller	Description
start	runtime	Signals init process that the container entrypoint should be executed
ready	init	Signals cmd process that the container creation is finished and the init process is waiting for a start signal
map_gid	init	Requests that cmd process writes gid mapping into procfs (contains variables)
map_uid	init	Requests that cmd process writes uid mapping into procfs (contains variables)

Table 5.4.: list of all the messages send between the *cmd process* and the *init process*

From table 5.4 a list of requirements can be created:

1. *cmd process* needs to send messages to *init process*
2. *init process* needs to send messages to *cmd process*
3. message must be able to contain variables
4. the communication needs to be synchronous

Item 4 is not necessary derived from the table of messages, it instead comes from the fact that both the *cmd process* and *init process* are in parts nondeterministic when exactly they will be ready to communicate with each other.

In addition it is to note that none of the messages expect a return value.

To meet these requirements a very minimal Remote Procedure Call (RPC) system is created, this system will fire and forget each procedure calls. This means that no return values or even a receive confirmation will be returned. This is possible not just because none of the messages expect an answer, but also because an error in either process leads to an abort of the complete operation, ending in the same result no matter where the error occurred.

The next section will explain the transport using Linux ipc features, the how the

messages are encoded for the transport and finally how the RPC system is implemented on either side.

5.6.1. Transport

There are a few different IPC features in Linux that meet these requirements like a Unix domain socket (uds), but considering how simple the messages between the processes are a more simple approach can be chosen: Named pipes (cf. 2.2.1).

Named pipe have two important properties that lead to the decision to base the message transport on them.

1. They are very efficient

2. By default, they only work when both reader and writer have opened the fd.

The second property solves the synchronization as it will basically pause reader or writer when either of the processes is not ready for communication.

Messages

Because the IPC is based on a very simple serial byte connection a wire format is needed to encode and decode the messages. For the message format itself protobufs (cf. 5.1.2) are used. These provide a simple IDL to define data structures and use code generation to provide a simple developer experience.

Each message defined in table 5.4 will be defined as a protobuf message. Messages without any associated values like *ready* will be defined as empty messages, messages that contain variables like *map_gid* will have these variables defined as fields inside the protobuf message (e.g. 5.17).

```
1 message IdMapping {
2  uint32 insideId = 2;
3  uint32 outsideId = 3;
4 }
```

Figure 5.17.: Example protobuf message

To easily differentiate between the messages coming from the same source, like ready and *map_gid* all messages will be encapsulated by in another message: From[Runtime|Init]. This message will be structured like the message in 5.18. Each source will get its own encapsulated message (FromRuntime and FromInit) and will contain one field named payload of the special protobuf type `oneof`. This special type enables the assignment of one of the defined messages inside of it during encoding. This provides a strongly typed message with flexible payloads.

```
message FromInit {
 oneof payload {
  Ready ready = 1;
  IdMapping map_gid = 2;
  IdMapping map_uid = 3;
 }
}
```

Figure 5.18.: Encapsulating protobuf message

The messages are then encoded into byte format, written to the fd of the named pipe and read by the reading process and then decoded again into the protobuf struct.

5.6.2. Connection

cmd process The cmd process creates the pipes during the `create` operation and before the *init process* is started. The pipes are created by the `createPipe` function (see 5.19) using the `mkfifo` syscall (cf. 2.2.1).

```
func createPipe(stateDir, pipeName string) error {
   fifoPath := filepath.Join(stateDir, pipeName)
   return syscall.Mkfifo(fifoPath, 0666)
}

func newPipeReader(stateDir, pipeName string) (*os.File, error) {
   fifoPath := filepath.Join(stateDir, pipeName)
   return os.OpenFile(fifoPath, os.O_RDONLY, os.ModeNamedPipe)
}

func newPipeWriter(stateDir, pipeName string) (*os.File, error) {
   fifoPath := filepath.Join(stateDir, pipeName)
   return os.OpenFile(fifoPath, os.O_WRONLY, os.ModeNamedPipe)
}
```

Figure 5.19.: Create and open pipes

The figure 5.19 shows the two similar functions `newPipeReader` and `newPipeWriter` that differ in the flag and mode used to open the file. Readers open the file read-only

and writers write-only, important is that both use the mode `NamedPipe` to access the fd of the named pipes created by `createPipe` (see 5.19).

5.6.3. RPC

With the transport logic done, the next level is the domain logic. The messages defined in 5.4 need appropriate implementations on either side to send messages and handle messages.

Two pipes will be created `runtime.pipe` for the *cmd process* and `init.pipe` for the *init process*.

runtime.pipe

The *cmd process* can receive 3 different messages defined in 5.4, 2 of them are requests to map ids in procfs using the idmapper 5.7.2 and one message is the ready signal.

The **reader of the runtime pipe** will listen to the incoming messages asynchronously using a goroutine (cf. [MBc]) and return a receive-only channel (cf. [MBb]).

The receive-only channel will receive an empty struct when the ready signal was received by the *init process* (cf. 5.5.1). The incoming mapping requests will be handled by the idmapper implementation of procfs.

The **writer of the runtime pipe** used by the *init process* will have methods to send these messages. The methods to send the mapping requests will implement the IdMapper interface (cf. 5.7.2) to enable a smooth developer experience abstracting away the IPC.

init.pipe

The *init process* can only receive a single message, the start signal. Because this is a single message type that is transmitted once no concurrency is needed to wait for this signal.

```
func (i *InitPipe) WaitForStart() error {
   ch := readFromRuntime(context.TODO(), i.fd)
   for msg := range ch {
      switch msg.Payload.(type) {
      case *pb.FromRuntime_Start:
         return nil
      default:
         return fmt.Errorf("unknown message")
      }
   }
   return fmt.Errorf("pipe closed without start signal")
}
```

Figure 5.20.: WaitForStart

The WaitForStart method (see 5.20) blocks until a message is received and only if its the start message returns without error.

The init pipe writer has one method to send the protobuf start message.

Lifecycle

Figure 5.21 shows the use of the pipes during the create and start operation of the container lifecycle (cf. 5.5). One thing that is not part of the diagram but still important is the *init process* is creating the contaienr isolation between *open init.pipe* and *send idmapping,* which is the reason that both pipes are opened before because it is not possible to open the fd of the *init pipe* after the process was isolated.

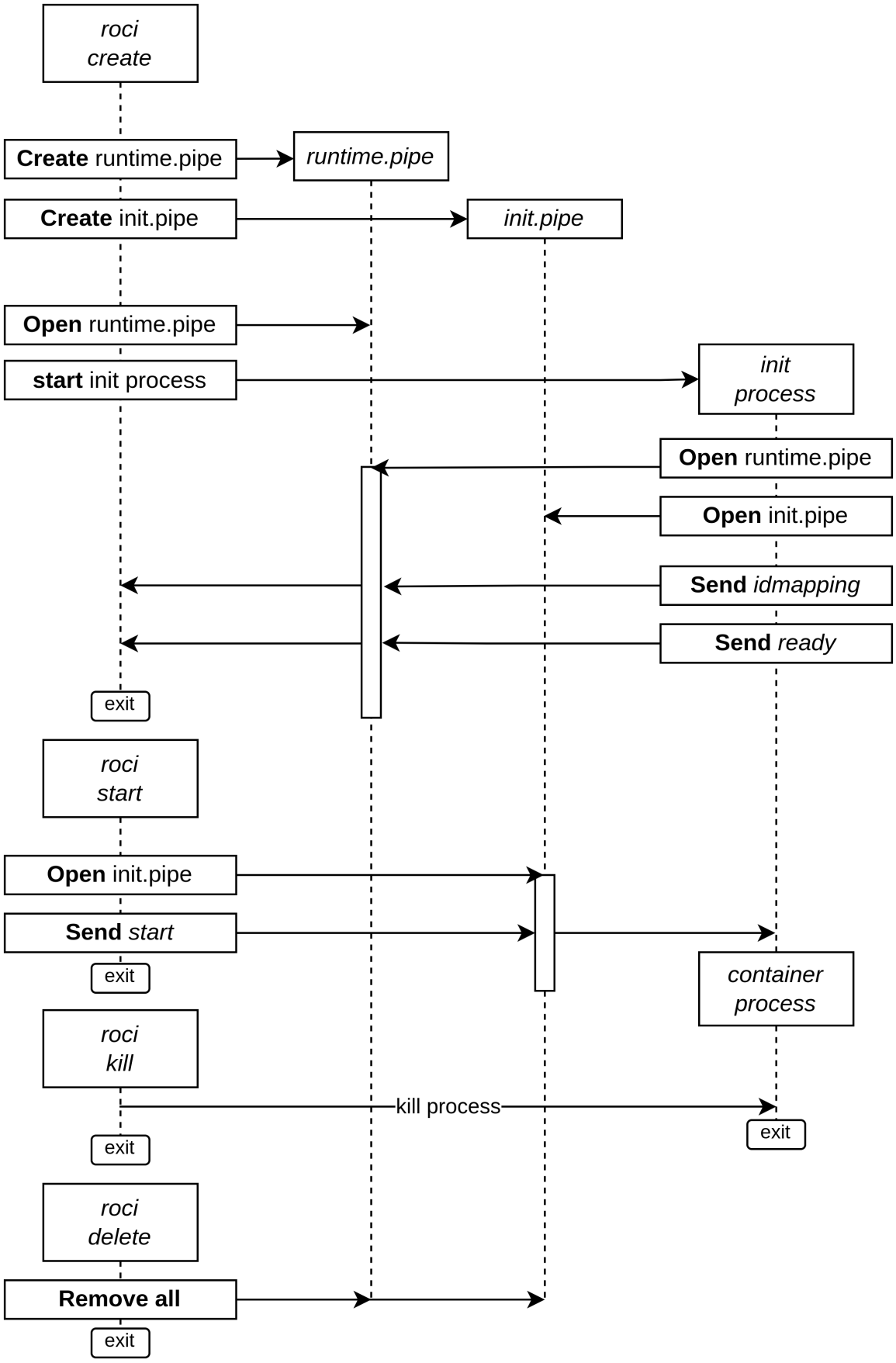

Figure 5.21.: IPC Communication sequence

5.7. Implementation of Containerization

This section will describe the implementation of the *init process* and its construction of the container isolation.

5.7.1. Init Process

The *init process* is responsible for building the container, it does this building isolating itself, preparing the rootfs and starting executing container entrypoint when ready.

When the *init process* is requested as described in Section 5.3.1 the *cmd process* uses the function in figure 5.22 to launch the process.

The `SysProcAttr` field of `cmd` allows the customization of the behavior of the process at a lower level. The `prepareCmd` uses to fields of `SysProcAttr` (see line 11 & 12 in 5.22). `Setsid` behaves like the syscall of the same name described in Section 2.2.1. The CloneFlags field is used to provide flags to the `clone` syscall (cf. 2.2.1), similar to the clone function if the *Standard C library* in figure 2.1. Why the `CLONE_NEWPID` flag is used is described in 5.7.2.

```
1  func prepareCmd(stateDir string) (*exec.Cmd, error) {
2     executablePath, err := os.Executable()
3     if err != nil {
4        return nil, err
5     }
6
7     cmd := exec.Command(executablePath, "init", stateDir)
8     cmd.Stdout = os.Stdout
9     cmd.Stderr = os.Stderr
10    cmd.SysProcAttr = &syscall.SysProcAttr{
11       Setsid: true,
12       Cloneflags: syscall.CLONE_NEWPID,
13    }
14    return cmd, nil
15 }
```

Figure 5.22.: Prepare init process

After the *init process* is launched it invokes the `Init` function of the `initp` package in `libcontainer`. This function then creates the namespaces (see 5.7.2), finalizes the rootfs (see 5.7.3), waits for the start signal (see 5.6.3) and then executes the container process (see 5.7.4).

5.7.2. Namespaces

The container configuration contains a list of namespaces that need to be created or assigned to the container. Namespaces that are not in the list should be inherited from the ancestor namespaces. Each entry in the list has the required field *type* and can have the optional field *path* (see 5.23). The *type* field is the name for a type of namespace. [Ope24c, Linux Container Configuration]

Some namespaces are not supported by roci (see 4.3.6), and in those cases the OCI runtime specification defines that the runtime must return an error. But to increase compatibility roci instead treat it as a warning.

The *path* field (see 5.23) can exist to specify an existing namespace that should be used instead of creating a new one. Joining existing namespaces is not implemented in roci and will result in an error.

```
1    "namespaces": [
2     {
3      "type": "pid"
4     },
5     {
6      "type": "network",
7      "path": "/run/netns/netns-7a297fda-02e1-6d4c-7995-d3c199f1c478"
8     },
9     ...
10   ]
```

Figure 5.23.: Partial namespaces list of a container configuration (cf. 2.3.1)

Each namespace defined in the OCI runtime specification [Ope24c] has a struct implementing the Namespace interface (see 5.24). This interface is used to control the containerization via namespaces. The interface contains 4 methods that provide metadata, and one method that implements necessary additional steps that are needed to finalize the containerization specific to that namespace.

```
1 type Namespace interface {
2    Priority() int
3    Type() specs.LinuxNamespaceType
4    CloneFlag() uintptr
5    IsSupported() bool
6
7     Finalize(spec specs.Spec) error
8 }
```

Figure 5.24.: libcontainer namespace interface

The *init process* than creates a list of namespaces from the OCI container configuration. This list is sorted by priority (see 5.24), which is a constant value set in the runtime implementation that determines the order of containerization during the creation of the container. This is necessary because some namespace require that either the creation of the namespace or the finalization of the namespace happens before or after other namespaces.

Then the namespaces are processed each namespace being unshared (using the unshare syscall, rf. 2.2.1) and then finalized using the `Finalize` method of the namespace interface (see 5.24).

The cgroup, mount, ipc (cf. 2.2.4) and time namespaces don´t have a finalize step. The other namespaces are explained in further details in the next sections:

Network

As specified in 4.3.6 the runtime does not implement the isolation of the network stack. The namespace is not unshared and no finalization steps are executed.

User

The user namespace (cf. 2.6) has the highest priority because if the user namespace is specified, all other namespaces should inherit from this user namespace when created. To finalize this namespace the id mapping described in 2.2.4 needs to happen.

To do that it uses the `IdMapper` (see 5.25) which is implemented in the procfs package. The `IdMapper` implements the mapping described in 2.6.

```
1 type IdMapper interface {
2    MapUid(pid Pid, insideId, outsideId uint32) error
3    MapGid(pid Pid, insideId, outsideId uint32) error
4 }
```

Figure 5.25.: IdMapper interface

But the mapping should be done by the *cmd process* because the *init process* is already in a partially isolated environment. To enable this, the namespace implementation does not use the idmapper implementation of the procfs package but a adapter that uses

the IPC described in 5.6 to pass the id mapping to the *cmd process* where the procfs implementation is then used.

PID

The PID namespace (cf. 2.2.4) is special as it is not unshared together with the other namespaces in the *init process* but is via clone flags as part of the *init process* start (see 5.22).

To finalize the containerization of the PID namespace the procfs (cf. 2.2.2) needs to be mounted inside the rootfs of the container, but this is not done in the finalize step of the namesapce but later during the finalization of the rootfs when all mounts specified for the rootfs are performed.

UTS

The UTS namespace (cf. 2.2.4) does not have a special priority, it is unshared in the init process and during its finalization the hostname and domain name specified in the container configuration are set.

5.7.3. Rootfs

The rootfs is the filesystem that is used as the filesystem of the container, which contains the files of the container image. It is supplied by the filesystem bundle (cf. 4.3.2) and its location is specified in the container configuration.

The OCI container configuration defines in multiple section steps that have to be implemented by the runtime to finalize the rootfs supplied by the filesystem bundle before it can be used as the container filesystem.

This implementation will ignore most of these steps except with two exceptions. The mounts will be implemented as specified in 4.3.5 and stdin, stdout and stderr will be attached to the rootfs via symlinks.

Finalize

After the namespaces are created, the rootfs is finalized.

Starting with mounting all of the mounts that are specified in the container configuration using the function shown in figure 5.26. This also includes the procfs, which is finalizing the containerization of the PID namespace (see 5.7.2).

```
func mountInRootfs(rootfs string, mount specs.Mount) (err error) {
   destination := filepath.Join(rootfs, mount.Destination)
   if err := os.MkdirAll(destination, 0o755); err != nil {
      return err
   }

   flags, data := oci.MountOptions(&mount)
   return syscall.Mount(mount.Source, destination, mount.Type, uintptr(flags), data)
}
```

Figure 5.26.: Mount in rootfs

After the mounting is finished the view of the container is restricted to the rootfs with chroot (cf. 2.2.1).

And finally the device symlinks are created with the function shown in figure 5.27 to attach stdin, stdout and stderr to the rootfs.

```
func createDevSymlinks(rootfs string) error {
   for _, link := range SpecDevSymlinks {
      var (
         source = link.Source
         target = filepath.Join(rootfs, link.Target)
      )
      if err := os.Symlink(source, target); err != nil && !os.IsExist(err) {
         return err
      }
   }

   return nil
}
```

Figure 5.27.: Create device symlinks

5.7.4. Entrypoint

The container's entrypoint is defined in [Ope24c, Configuration, Process] with the two fields of the *process* object:

> - **env** (array of strings, OPTIONAL) with the same semantics as IEEE Std 1003.1-2008's environ.
> - **args** (array of strings, OPTIONAL) with similar semantics to IEEE Std 1003.1-2008 execvp's argv. [...]
>
> - [Ope24c, Configuration, Process]

The parameters of the `exec` syscall are prepared by the `Entrypoint` function (see 5.28. The first argument of the `args` array is assumed to be the executable and is looked up inside the container environment.

The `exec` syscall then tries to execute the executable, it will retry if the syscall is interrupted and if it succeeds it will be the end of the *init process* because it is now the container process.

```
1  func Entrypoint(process *specs.Process) (string, []string, []string, error) {
2     name, err := exec.LookPath(process.Args[0])
3     if err != nil {
4        return "", nil, nil, err
5     }
6     process.Args[0] = name
7     return process.Args[0], process.Args, process.Env, nil
8  }
9
10 func execEntrypoint(spec specs.Spec) (err error) {
11    arg0, args, env, err := Entrypoint(spec.Process)
12    if err != nil {
13       return err
14    }
15
16    for {
17       err = syscall.Exec(arg0, args, env)
18       if !errors.Is(err, syscall.EINTR) {
19          return err
20       }
21    }
22 }
```

Figure 5.28.: Execute container entrypoint

6

Validation

6.1. Test

In addition to unit tests defined in code the runtime operations to manage the lifecycle of a container are manually tested. These manual tests also depend on external dependency like a ready filesystem bundle and installed Podman.

6.1.1. Manual Testing

To verify that roci is able to perform the standard operations defined in the OCI runtime specification and is compatible with the designated runtime caller (cf. 4.3.1) Podman will be defined.

Tests

Table 6.1 shows a list of tests that should be performed manually to verify that the container lifecycle management is supported by roci and that it is compatible with Podman.

These tests were performed and confirmed that roci is working as expected with one exception. Step 4 in 6.1 specifies that Podman should be used to kill the container. This is not possible because roci is not behaving as Podman expects, but killing the container directly using roci works as expected.

Step	Runtime Operations	Command	Description	Expected Output
1	None	`sudo podman create --runtime roci alpine sleep 100`	This step is preparation for podman, none of the container operations implemented by roci are used in this step	containerid
2	**CREATE,START**	`sudo podman start <containerId form step 1>`	In this step the container is started by podman. Podman uses roci to create and start the container	containerId
3	**STATE**	`sudo roci state <containerId from step 1>`	This step confirms that the container is managed by roci and tests the output of the state command	see 5.10
4	**KILL,DELETE**	`sudo podman kill <containerId>`	In this step the running container is killed by podman using roci.	containerId
5	**STATE**	`sudo roci state <containerId from step 1>`	Podman will use the delete operation as soon as the container process is stopped	“Error: container does not exist”

Table 6.1.: Manual tests

6.2. Benchmark

Benchmarking roci and other container runtimes is crucial to understand the performance characteristics of these runtimes and how much potential overhead was reduced by roci. This section presents a comparison of roci against two OCI-compliant runtimes and will measure their performance of the basic container operations: create, start, delete.

Benchmarking container runtimes is crucial for understanding their performance characteristics, particularly in environments where efficiency and speed are paramount. This section presents a comparison of three different OCI-compliant runtimes. Each runtime was evaluated using the hyperfine benchmarking tool, with the specific command sequence focused on container lifecycle management—creation, starting, and deletion.

6.2.1. Methodology

The methodology is based on benchmarks that contributors of the youki container runtime published in their readme document [you24b].

Each runtime was evaluated using the *hyperfine* benchmark tool (cf. [Pet23]), with a specific sequence of commands going through the basic operations of the container lifecycle management.

Benchmark Command

```
hyperfine \
--prepare 'sudo sync; echo 3 | sudo tee /proc/sys/vm/drop_caches' \
--warmup 10 \
--min-runs 100 \
"sudo $RUNTIME_EXECUTABLE create -b /tmp/roci a && sudo $RUNTIME_EXECUTABLE
   start a && sudo $RUNTIME_EXECUTABLE delete -f a"
```

Figure 6.1.: Benchmark command

The hyperfine command in figure 6.1 has 4 parts to it.

Cache Clearing Before each benchmark run, system caches were cleared using `sync` and `echo 3 | sudo tee /proc/sys/vm/drop\_caches` to reduce the impact of previous runs on the current benchmark.

Warmup Phase Each runtime was warmed up with 10 initial executions to prime the system and avoid cold start anomalies.

Repetition A minimum of 100 runs per runtime was performed to obtain statistically significant results.

Command to benchmark The tested command is a combination of three lifecycle steps, create, start and delete.

The filesystem bundle (cf. 4.3.2) required for **create** is created in three steps.

1. Create rootfs
2. Create container configuration
3. Modifiy container configuration

Because roci does not implement many of the requirements from the OCI runtime specification the choice of the rootfs is a bit more restrictive. Mainly a rootfs is required that contains a Linux filesystem that is largely complete (e.g. Default filesystem exists, filesystem is of common Linux structure, etc). For that the alpine image 23.0.11 was selected (see A.2.2).

The image is exported (using: `podman export <containerid> > rootfs.tar`) and extracted into the benchmark directory (using: `tar -xf rootfs.tar`). With that step 1 is finished.

For step 2 a container configuration has to be created. The container runtime runc implements a command to generate a container configuration based on a rootfs with the command `runc spec`. The generated config is attached at A.2.1.

Now to finish with step 3 the container configuration has to be modified. Line 2 of A.2.1 has to be removed. If it is not removed it would cause OCI compliant runtimes to

require additional configuration and roci does not support this configuration.

Now the filesystem bundle for the benchmark is created.

Benchmark Environment

The benchmark is run on EndeavourOS with the Linux Kernel 6.10.5-arch1-1.

- Memory: 7662MiB
- CPU: 11th Gen Intel i5-1135G7 (8) @ 4.200GHz
- CPU Architecture: x86_64

Runtime Selection

Three runtimes are going the be tested and compared.

- roci
- runc, because it is the most common runtime and implemented in the same language as roci
- crun, because it is optimized for performance and one of the fastest container runtimes.

6.2.2. Results

Runtime	Time (mean ±)	Range (min … max)	vs roci (mean)	Version
roci	61.3 ms ± 1.0 ms	57.3 ms … 65.3 ms	100%	
runc	102.4 ms ± 11.6 ms	77.9 ms … 139.0 ms	+67.04%	1.1.13
crun	48.8 ms ± 3.1 ms	44.9 ms … 67.2 ms	−20.40%	1.16.1

Table 6.2.: Benchmark results

roci being the runtime implemented in this thesis served as the baseline for comparison. It had a mean execution time of 61.3 ms and, compared to the other implementations, a remarkably low standard deviation of 1.0 ms. Its execution times are tightly clustered between 57.3 ms and 65.3 ms, indicating consistent and predictable performance.

runc is the reference implementation of the OCI runtime specification and like roci was written in Go, which makes the overhead more directly comparable. It the highest mean execution time at 102.4 ms, with a significant standard deviation of 11.6 ms. The wide range of execution times (77.9 ms to 139.0 ms) suggests that runc is prone to variability.

Compared to roci is runc 67.04% slower. This does indicate that a lot of overhead can be reduced by selecting which features defined in the OCI runtime specification to implement. An interesting side effect is also that the performance of roci seems to be a lot more consistent compared to runc's performance with a standard deviation of 1ms in roci's results and a standard deviation of 11.6ms in runc's results.

crun But crun still outperforms both roci and runc, with a mean execution time of 48.8 ms. The range of 44.9 ms to 67.2 ms, while broader than roci, still indicates a fairly consistent runtime.

This means that crun is still 20.40% faster than roci. Its performance advantage likely stems from its implementation in C.

7

Summary

7.1. Conclusion

This thesis aimed to describe what makes containerization possible, to gain insight into the OCI runtime specification and to implement a minimal version of an OCI runtime.

While this thesis has been able to provide detailed insight into cgroups and especially namespaces which are fundamental building blocks of containerization, it must said that the OCI container defined by the OCI runtime specification goes far beyond those fundamental features. And many of these features were either not described at all, or not implemented in the runtime.

Furthermore the implementation of the OCI runtime specification, even with the reduced scope described in Chapter 4, has proven itself to be far more work than initially expected. Most of the additional work can be traced back to two things.

The OCI runtime specification does not define a API and there is no standardized API definition by the OCI at all. Instead, the runtimes that implement the OCI runtime specification are copying the runc CLI to provide interoperability with runtime callers, which also required more exploratory research than initially expected because there is very little documentation besides the help output of runc itself.

Additional work was also caused by the specification itself. While it describes what is required to containerize processes, it does very little to specify how. Which also turned out to be a topic that needed a lot of exploratory research. In particular, to understand how to implement a containerization process that is compliant with the operations defined in the OCI runtime specification.

Also, the goal of preserving container portability was not fully achieved, as the missing

network isolation reduces portability because the process inside the container has to be aware of network restrictions caused by the host or other containers, such as ports in use by other processes.

However, the benchmark has shown that this minimal implementation of the OCI runtime specification reduces the overhead and significantly increases its performance compared to the reference implementation runc. Another insight gained from the results of the benchmark is that this minimal implementation presents more consistent performance characteristics than runc and crun.

But the results also show that crun, that implements the complete OCI specification, presents better performance than the minimal runtime roci.

In conclusion, the implementation of the OCI runtime specification turned out to be more exploratory than expected, but a minimal runtime that is partially interoperable with a standard runtime caller was implemented and outperformed the reference implementation of the OCI runtime.

7.2. Outlook

The related work has already shown that there are many different approaches to containerization for embedded systems. It will be interesting to see if a single approach will reach a similar market share as Docker/OCI did in the cloud space.

The implementation in this thesis left large parts of the OCI runtime specification untouched, leaving much potential for further work. The biggest gain in performance can be achieved by implementing the runtime in a language like C to gain the advantages visible in the benchmark results. Also, a more complete implementation of the OCI runtime specification could provide further insight into the performance characteristics of OCI container runtimes.

In light of how complex the implementation of the OCI runtime specification turned out to be there is also a different direction that could be considered. Rather than implementing the specification from scratch one could start with an existing implementation like runc and remove parts of it to gain more meaningful insight into resource overhead caused by runtimes.

7.3. Open Source

The implementation that was described in this thesis is available on Github and is published under the MIT License at **`github.com/m4schini/roci`**.

A

Attachments

A.1. runc command line documentation

A.1.1. runc help

```
NAME:
   runc - Open Container Initiative runtime

runc is a command line client for running applications packaged according to
the Open Container Initiative (OCI) format and is a compliant implementation of the
Open Container Initiative specification.

runc integrates well with existing process supervisors to provide a production
container runtime environment for applications. It can be used with your
existing process monitoring tools and the container will be spawned as a
direct child of the process supervisor.

Containers are configured using bundles. A bundle for a container is a directory
that includes a specification file named "config.json" and a root filesystem.
The root filesystem contains the contents of the container.

To start a new instance of a container:

    # runc run [ -b bundle ] <container-id>

Where "<container-id>" is your name for the instance of the container that you
are starting. The name you provide for the container instance must be unique on
your host. Providing the bundle directory using "-b" is optional. The default
value for "bundle" is the current directory.

USAGE:
   runc [global options] command [command options] [arguments...]

VERSION:
   1.1.12-0ubuntu2~22.04.1
spec: 1.0.2-dev
go: go1.21.1
libseccomp: 2.5.3

COMMANDS:
   checkpoint  checkpoint a running container
   create      create a container
   delete      delete any resources held by the container often used with detached container
   events      display container events such as OOM notifications, cpu, memory, and IO usage statistics
   exec        execute new process inside the container
   kill        kill sends the specified signal (default: SIGTERM) to the container's init process
   list        lists containers started by runc with the given root
   pause       pause suspends all processes inside the container
   ps          ps displays the processes running inside a container
   restore     restore a container from a previous checkpoint
   resume      resumes all processes that have been previously paused
   run         create and run a container
   spec        create a new specification file
   start       executes the user defined process in a created container
   state       output the state of a container
   update      update container resource constraints
   features    show the enabled features
   help, h     Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --debug               enable debug logging
   --log value           set the log file to write runc logs to (default is '/dev/stderr')
   --log-format value    set the log format ('text' (default), or 'json') (default: "text")
   --root value          root directory for storage of container state (this should be located in tmpfs) (default: "/run/user/1000//runc")
   --criu value          path to the criu binary used for checkpoint and restore (default: "criu")
   --systemd-cgroup      enable systemd cgroup support, expects cgroupsPath to be of form "slice:prefix:name" for e.g. "system.slice:runc:434234"
   --rootless value      ignore cgroup permission errors ('true', 'false', or 'auto') (default: "auto")
   --help, -h            show help
   --version, -v         print the version
```

Output of "runc help"

A.1.2. runc create

```
NAME:
   runc create — create a container

USAGE:
   runc create [command options] <container-id>

Where "<container-id>" is your name for the instance of the container that you
are starting. The name you provide for the container instance must be unique on
your host.

DESCRIPTION:
   The create command creates an instance of a container for a bundle. The bundle
is a directory with a specification file named "config.json" and a root
filesystem.

The specification file includes an args parameter. The args parameter is used
to specify command(s) that get run when the container is started. To change the
command(s) that get executed on start, edit the args parameter of the spec. See
"runc spec --help" for more explanation.

OPTIONS:
   --bundle value, -b value  path to the root of the bundle directory, defaults to the current directory
   --console-socket value    path to an AF_UNIX socket which will receive a file descriptor referencing the master end of the console's pseudoterminal
   --pid-file value          specify the file to write the process id to
   --no-pivot                do not use pivot root to jail process inside rootfs.  This should be used whenever the rootfs is on top of a ramdisk
   --no-new-keyring          do not create a new session keyring for the container.  This will cause the container to inherit the calling processes session key
   --preserve-fds value      Pass N additional file descriptors to the container (stdio + $LISTEN_FDS + N in total) (default: 0)
```

Output of “runc create -h”

A.1.3. runc start

```
NAME:
   runc start — executes the user defined process in a created container

USAGE:
   runc start <container-id>

Where "<container-id>" is your name for the instance of the container that you
are starting. The name you provide for the container instance must be unique on
your host.

DESCRIPTION:
   The start command executes the user defined process in a created container.
```

Output of “runc start -h”

A.1.4. runc kill

```
NAME:
   runc kill — kill sends the specified signal (default: SIGTERM) to the container's init process

USAGE:
   runc kill [command options] <container-id> [signal]

Where "<container-id>" is the name for the instance of the container and
"[signal]" is the signal to be sent to the init process.

EXAMPLE:
For example, if the container id is "ubuntu01" the following will send a "KILL"
signal to the init process of the "ubuntu01" container:

       # runc kill ubuntu01 KILL

OPTIONS:
   --all, -a  send the specified signal to all processes inside the container
```

Output of "runc kill -h"

A.1.5. runc delete

```
NAME:
   runc delete — delete any resources held by the container often used with detached container

USAGE:
   runc delete [command options] <container-id>

Where "<container-id>" is the name for the instance of the container.

EXAMPLE:
For example, if the container id is "ubuntu01" and runc list currently shows the
status of "ubuntu01" as "stopped" the following will delete resources held for
"ubuntu01" removing "ubuntu01" from the runc list of containers:

       # runc delete ubuntu01

OPTIONS:
   --force, -f  Forcibly deletes the container if it is still running (uses SIGKILL)
```

Output of "runc delete -h"

A.1.6. runc state

```
NAME:
   runc state — output the state of a container

USAGE:
   runc state <container-id>

Where "<container-id>" is your name for the instance of the container.

DESCRIPTION:
   The state command outputs current state information for the
instance of a container.
```

Output of "runc state -h"

A.2. Benchmark

A.2.1. container configuration generated by runc

```
{
    "ociVersion": "1.0.2-dev",
    "console": true
    "process": {
        "user": {
            "uid": 0,
            "gid": 0
        },
        "args": [
            "sh"
        ],
        "env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
            "TERM=xterm"
        ],
        "cwd": "/",
        "capabilities": {
            "bounding": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL",
                "CAP_NET_BIND_SERVICE"
            ],
            "effective": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL",
                "CAP_NET_BIND_SERVICE"
            ],
            "permitted": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL",
                "CAP_NET_BIND_SERVICE"
            ],
            "ambient": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL",
                "CAP_NET_BIND_SERVICE"
            ]
        },
        "rlimits": [
            {
                "type": "RLIMIT_NOFILE",
                "hard": 1024,
                "soft": 1024
            }
        ],
        "noNewPrivileges": true
    },
    "root": {
        "path": "rootfs",
        "readonly": true
    },
    "hostname": "runc",
     "mounts": [
        {
            "destination": "/proc",
            "type": "proc",
            "source": "proc"
        },
        {
            "destination": "/dev",
            "type": "tmpfs",
            "source": "tmpfs",
            "options": [
                "nosuid",
                "strictatime",
                "mode=755",
                "size=65536k"
            ]
        },
        {
            "destination": "/dev/pts",
            "type": "devpts",
            "source": "devpts",
            "options": [
                "nosuid",
                "noexec",
                "newinstance",
                "ptmxmode=0666",
                "mode=0620",
                "gid=5"
            ]
        },
        {
            "destination": "/dev/shm",
            "type": "tmpfs",
            "source": "shm",
            "options": [
                "nosuid",
                "noexec",
                "nodev",
                "mode=1777",
                "size=65536k"
            ]
        },
        {
            "destination": "/dev/mqueue",
            "type": "mqueue",
            "source": "mqueue",
            "options": [
                "nosuid",
                "noexec",
                "nodev"
            ]
        },
        {
            "destination": "/sys",
            "type": "sysfs",
            "source": "sysfs",
            "options": [
                "nosuid",
                "noexec",
                "nodev",
                "ro"
            ]
        },
        {
            "destination": "/sys/fs/cgroup",
            "type": "cgroup",
            "source": "cgroup",
            "options": [
                "nosuid",
                "noexec",
                "nodev",
                "relatime",
                "ro"
            ]
        }
    ],
    "linux": {
        "resources": {
            "devices": [
                {
                    "allow": false ,
                    "access": "rwm"
                }
            ]
        },
        "namespaces": [
            {
                "type": "pid"
            },
            {
                "type": "network"
            },
            {
                "type": "ipc"
            },
            {
                "type": "uts"
            },
            {
                "type": "mount"
            },
            {
                "type": "cgroup"
            }
        ],
        "maskedPaths": [
            "/proc/acpi",
            "/proc/asound",
            "/proc/kcore",
            "/proc/keys",
            "/proc/latency_stats",
            "/proc/timer_list",
            "/proc/timer_stats",
            "/proc/sched_debug",
            "/sys/firmware",
            "/proc/scsi"
        ],
        "readonlyPaths": [
            "/proc/bus",
            "/proc/fs",
            "/proc/irq",
            "/proc/sys",
            "/proc/sysrq-trigger"
        ]
    }
}
```

A.2.2. Alpine container image


```
[
    {
        "Id": "a606584aa9aa875552092ec9e1d62cb98d486f51f389609914039aabd9414687",
        "Digest": "sha256:dabf91b69c191a1a0a1628fd6bdd029c0c4018041c7f052870bb13c5a222ae76",
        "RepoTags": [
            "docker.io/library/alpine:latest"
        ],
        "RepoDigests": [
            "docker.io/library/alpine@sha256:b89d9c93e9ed3597455c90a0b88a8bbb5cb7188438f70953fede212a0c4394e0",
            "docker.io/library/alpine@sha256:dabf91b69c191a1a0a1628fd6bdd029c0c4018041c7f052870bb13c5a222ae76"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2024-06-20T20:16:58.064410339Z",
        "Config": {
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh"
            ]
        },
        "Version": "23.0.11",
        "Author": "",
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 8087324,
        "VirtualSize": 8087324,
        "GraphDriver": {
            "Name": "overlay",
            "Data": {
                "UpperDir": "/home/m4schini/.local/share/containers/storage/overlay/94
                    e5f06ff8e3d4441dc3cd8b090ff38dc911bfa8ebdb0dc28395bc98f82f983f/diff",
                "WorkDir": "/home/m4schini/.local/share/containers/storage/overlay/94
                    e5f06ff8e3d4441dc3cd8b090ff38dc911bfa8ebdb0dc28395bc98f82f983f/work"
            }
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:94e5f06ff8e3d4441dc3cd8b090ff38dc911bfa8ebdb0dc28395bc98f82f983f"
            ]
        },
        "Labels": null,
        "Annotations": {},
        "ManifestType": "application/vnd.docker.distribution.manifest.v2+json",
        "User": "",
        "History": [
            {
                "created": "2024-06-20T20:16:57.940229876Z",
                "created_by": "/bin/sh -c #(nop) ADD file:33ebe56b967747a97dcec01bc2559962bee8823686c9739d26be060381bbb3ca in / "
            },
            {
                "created": "2024-06-20T20:16:58.064410339Z",
                "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",
                "empty_layer": true
            }
        ],
        "NamesHistory": [
            "docker.io/library/alpine:latest"
        ]
    }
]
```


alpine image metadata

Bibliography

[Acc24] Accenture-Industry X. *esrlabs/northstar*. original-date: 2020-06-29T16:25:24Z. June 4, 2024. url: https://github.com/esrlabs/northstar (visited on 06/07/2024).

[Ant19] Anton. *Understanding Docker container escapes*. Trail of Bits Blog. July 20, 2019. url: https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/ (visited on 08/17/2024).

[Bel23] Adam Gordon Bell. *Containers are chroot with a Marketing Budget - Earthly Blog*. Earthly. July 19, 2023. url: https://earthly.dev/blog/chroot/ (visited on 08/14/2024).

[Bou22] Shlomi Boutnaru. *The Linux Process Journey — PID 1 (init)*. Medium. Aug. 17, 2022. url: https://medium.com/@boutnaru/the-linux-process-journey-pid-1-init-60765a069f17 (visited on 07/27/2024).

[Clo24] Cloud Native Computing Foundation. *CNCF Landscape*. 2024. url: https://landscape.cncf.io/?view-mode=grid (visited on 08/07/2024).

[cob24a] cobra contributers. *spf13/cobra*. original-date: 2013-09-03T20:40:26Z. July 9, 2024. url: https://github.com/spf13/cobra (visited on 07/09/2024).

[cob24b] cobra-cli contributers. *spf13/cobra-cli*. original-date: 2022-02-10T02:37:32Z. Aug. 14, 2024. url: https://github.com/spf13/cobra-cli (visited on 08/14/2024).

[cT24] containerd Authors and The Linux Foundation. *containerd*. 2024. url: https://containerd.io/ (visited on 07/25/2024).

[Flo97] Glen Flower. “Processes and Process Context”. In: *Linux Gazette* 23 (Dec. 1997). url: https://tldp.org/LDP/LG/issue23/flower/psimage.html (visited on 08/13/2024).

[git24] github.com/containers contributers. *crun*. original-date: 2017-09-13T20:20:58Z. July 30, 2024. url: https://github.com/containers/crun (visited on 07/31/2024).

[Goo24a] Google. *APEX file format*. Android Open Source Project. Aug. 1, 2024. url: https://source.android.com/docs/core/ota/apex (visited on 08/17/2024).

[Goo19] Google. *Go for Cloud & Network Services - The Go Programming Language*. Oct. 4, 2019. url: https://go.dev/solutions/cloud (visited on 08/14/2024).

[Goo24b] Google. *Protocol Buffers*. 2024. url: https://protobuf.dev/ (visited on 08/23/2024).

[Goo24c] Google. *protocolbuffers/protobuf-go*. In collab. with protocol buffer contributers. original-date: 2019-03-26T06:26:40Z. Aug. 23, 2024. url: https://github.com/protocolbuffers/protobuf-go (visited on 08/23/2024).

[Gra18] Stéphane Graber. *LXC 3.0.0 has been released - News*. Linux Containers Forum. Section: News. Mar. 27, 2018. url: https://discuss.linuxcontainers.org/t/lxc-3-0-0-has-been-released/1449 (visited on 08/13/2024).

[HR15] Tim Hildred and Red Hat Inc. *The History of Containers*. Red Hat Blog. Aug. 28, 2015. url: https://www.redhat.com/en/blog/history-containers (visited on 06/19/2024).

[Hyk13] Solomon Hykes. *Lightning Talk - The future of Linux Containers*. PyVideo.org. Mar. 15, 2013. url: https://pyvideo.org/pycon-us-2013/the-future-of-linux-containers.html (visited on 07/16/2024).

[HD15] Solomon Hykes and Docker Inc. *Introducing runC: a lightweight universal container runtime | Docker*. Section: Community. June 22, 2015. url: https://www.docker.com/blog/runc/ (visited on 06/25/2024).

[Iva17] Konstantin Ivanov. *Containerization with LXC : Get Acquainted with the World of LXC*. Packt Publishing, Feb. 2017. isbn: 978-1-78588-894-6. url: https://library.haeck.se/book/56 (visited on 06/19/2024).

[Lin24a] Linux man-pages project. *capabilities(7) - Linux manual page*. June 13, 2024. url: https://www.man7.org/linux/man-pages/man7/capabilities.7.html (visited on 08/17/2024).

[Lin24b] Linux man-pages project. *cgroups(7) - Linux manual page*. June 15, 2024. url: https://www.man7.org/linux/man-pages/man7/cgroups.7.html (visited on 07/31/2024).

[Lin24c] Linux man-pages project. *chroot(2) - Linux manual page*. May 2, 2024. url: https://www.man7.org/linux/man-pages/man2/chroot.2.html (visited on 08/13/2024).

[Lin24d] Linux man-pages project. *exec(3) - Linux manual page*. June 16, 2024. url: https://www.man7.org/linux/man-pages/man3/exec.3.html (visited on 08/09/2024).

[Lin24e] Linux man-pages project. *fork(2) - Linux manual page*. June 15, 2024. url: https://man7.org/linux/man-pages/man2/fork.2.html (visited on 08/13/2024).

[Lin24f] Linux man-pages project. *inode(7) - Linux manual page*. June 15, 2024. url: https://www.man7.org/linux/man-pages/man7/inode.7.html (visited on 08/20/2024).

[Lin24g] Linux man-pages project. *ipc_namespaces(7) - Linux manual page*. May 2, 2024. url: https://www.man7.org/linux/man-pages/man7/ipc_namespaces.7.html (visited on 07/29/2024).

[Lin24h] Linux man-pages project. *mkfifo(3) - Linux manual page*. May 2, 2024. url: https://www.man7.org/linux/man-pages/man3/mkfifo.3.html (visited on 08/13/2024).

[Lin24i] Linux man-pages project. *mount(2) - Linux manual page*. June 13, 2024. url: https://www.man7.org/linux/man-pages/man2/mount.2.html (visited on 08/13/2024).

[Lin24j] Linux man-pages project. *mount_namespaces(7) - Linux manual page*. June 15, 2024. url: https://www.man7.org/linux/man-pages/man7/mount_namespaces.7.html (visited on 07/29/2024).

[Lin24k] Linux man-pages project. *mq_overview(7) - Linux manual page*. May 2, 2024. url: https://www.man7.org/linux/man-pages/man7/mq_overview.7.html (visited on 08/17/2024).

[Lin24l] Linux man-pages project. *namespaces(7) - Linux manual page*. June 13, 2024. url: https://www.man7.org/linux/man-pages/man7/namespaces.7.html (visited on 07/25/2024).

[Lin24m] Linux man-pages project. *network_namespaces(7) - Linux manual page*. May 2, 2024. url: https://www.man7.org/linux/man-pages/man7/network_namespaces.7.html (visited on 07/29/2024).

[Lin24n] Linux man-pages project. *pid_namespaces(7) - Linux manual page*. June 13, 2024. url: https://www.man7.org/linux/man-pages/man7/pid_namespaces.7.html (visited on 07/29/2024).

[Lin24o] Linux man-pages project. *proc_pid(5) - Linux manual page*. May 2, 2024. url: https://man7.org/linux/man-pages/man5/proc_self.5.html (visited on 07/31/2024).

[Lin24p] Linux man-pages project. *proc_pid_mounts(5) - Linux manual page*. May 2, 2024. url: https://man7.org/linux/man-pages/man5/proc_pid_mounts.5.html (visited on 07/31/2024).

[Lin24q] Linux man-pages project. *syscalls(2) - Linux manual page*. May 2, 2024. url: https://man7.org/linux/man-pages/man2/syscalls.2.html (visited on 07/31/2024).

[Lin24r] Linux man-pages project. *sysvipc(7) - Linux manual page*. May 2, 2024. url: https://www.man7.org/linux/man-pages/man7/sysvipc.7.html (visited on 07/31/2024).

[Lin24s] Linux man-pages project. *unshare(2) - Linux manual page*. June 15, 2024. url: https://www.man7.org/linux/man-pages/man2/unshare.2.html (visited on 08/13/2024).

[Lin24t] Linux man-pages project. *user_namespaces(7) - Linux manual page*. June 15, 2024. url: https://www.man7.org/linux/man-pages/man7/user_namespaces.7.html (visited on 07/29/2024).

[Lin24u] Linux man-pages project. *uts_namespaces(7) - Linux manual page*. May 2, 2024. url: https://www.man7.org/linux/man-pages/man7/uts_namespaces.7.html (visited on 07/29/2024).

[LT15a] LSB Workgroup and The Linux Foundation. *Chapter 6. Operating System Specific Annex*. 2015. url: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/ch06.html#procKernelAndProcessInformationVir (visited on 07/29/2024).

[LT15b] LSB Workgroup and The Linux Foundation. *Filesystem Hierarchy Standard*. 2015. url: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html (visited on 06/28/2024).

[LXC24] LXC contributors. *Linux Containers - LXC - Introduction*. 2024. url: https://linuxcontainers.org/ (visited on 06/25/2024).

[MBa] Mark McGranaghan and Eli Bendersky. *Go by Example*. url: https://gobyexample.com/ (visited on 08/15/2024).

[MBb] Mark McGranaghan and Eli Bendersky. *Go by Example: Channels*. url: https://gobyexample.com/channels (visited on 08/15/2024).

[MBc] Mark McGranaghan and Eli Bendersky. *Go by Example: Goroutines*. url: https://gobyexample.com/goroutines (visited on 08/15/2024).

[Ope15] Open Container Initiative. *Industry Leaders Unite to Create Project for Open Container Standards - Open Container Initiative*. June 20, 2015. url: https://opencontainers.org/posts/announcements/2015-06-20-industry-leaders-unite-to-create-project-for-open-container-standard/ (visited on 07/16/2024).

[Ope24a] Open Container Initiative. *OCI Image Format Specification v1.1.0*. Feb. 15, 2024. url: https://github.com/opencontainers/image-spec/releases/tag/v1.1.0 (visited on 06/07/2024).

[Ope24b] Open Container Initiative. *Open Container Initiative Distribution Specification v1.1.0*. Feb. 15, 2024. url: https://github.com/opencontainers/distribution-spec/releases/download/v1.1.0/oci-distribution-spec-v1.1.0.pdf.

[Ope24c] Open Container Initiative. *Open Container Initiative Runtime Specification v1.2.0*. Feb. 13, 2024. url: https://github.com/opencontainers/runtime-spec/releases/tag/v1.2.0 (visited on 06/07/2024).

[Ope24d] Open Container Initiative. *opencontainers/artifacts*. original-date: 2019-08-16T22:14:38Z. July 18, 2024. url: https://raw.githubusercontent.com/opencontainers/artifacts/main/README.md (visited on 07/31/2024).

[Ope23] Open Container Initiative. *opencontainers/runtime-spec README*. June 23, 2023. url: https://raw.githubusercontent.com/opencontainers/runtime-spec/8f3fbc881602d85699e5c448634ec1288860d966/README.md (visited on 08/17/2024).

[Ope24e] Open Container Initiative. *runc*. In collab. with github.com/opencontainers contributers. original-date: 2015-06-05T23:30:45Z. July 9, 2024. url: https://github.com/opencontainers/runc (visited on 07/09/2024).

[Ove22] Steve Ovens. *Building containers by hand using namespaces: The net namespace*. Enable Sysadmin. Publisher: Red Hat, Inc. Section: Enable Sysadmin. Mar. 9, 2022. url: https://www.redhat.com/sysadmin/net-namespaces (visited on 05/09/2023).

[Pana] Pantacor. *Pantacor – Building and managing embedded Linux with DevOps tools*. url: https://pantacor.com/ (visited on 06/25/2024).

[Panb] Pantacor. *Pantavisor*. url: https://www.pantavisor.io (visited on 06/25/2024).

[Pan21] Pantacor. *pantavisor/README.md*. GitHub. Nov. 23, 2021. url: https://github.com/pantavisor/pantavisor/blob/9d160265dd7da1911ba6ec122b0fd34ee8README.md (visited on 08/19/2024).

[Pet23] David Peter. *hyperfine*. Version 1.16.1. original-date: 2018-01-13T15:49:54Z. Mar. 2023. url: https://github.com/sharkdp/hyperfine (visited on 08/22/2024).

[RM15] Red Hat Inc. and Scott McCarty. *Architecting Containers Part 1: Why Understanding User Space vs. Kernel Space Matters*. Red Hat Blog. July 29, 2015. url: https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-matters (visited on 06/19/2024).

[Rhe03] Rheinwerk Verlag. *Rheinwerk Computing: Wie werde ich UNIX-Guru? - Kapitel: Network Information Service: NIS*. Rheinwerk Computing Openbook. 2003. url: https://openbook.rheinwerk-verlag.de/unix_guru/node205.html (visited on 07/29/2024).

[Tay11] Tayyab. *Answer to "What are file descriptors, explained in simple terms?"* Stack Overflow. Mar. 10, 2011. url: https://stackoverflow.com/a/5256705 (visited on 08/17/2024).

[TC24] The Linux Foundation and Cloud Native Computing Foundation. *Cloud Native Computing Foundation Annual Survey 2023*. Apr. 9, 2024. url: https://www.cncf.io/reports/cncf-annual-survey-2023/ (visited on 06/19/2024).

[Ube24] Uber. *uber-go/zap*. In collab. with zap contributers. original-date: 2016-02-18T19:52:56Z. Aug. 23, 2024. url: https://github.com/uber-go/zap (visited on 08/23/2024).

[vip24] viper contributers. *spf13/viper*. original-date: 2014-04-02T14:33:33Z. July 9, 2024. url: https://github.com/spf13/viper (visited on 07/09/2024).

[W3C15] W3Cook. *OS Usage Trends and Market Share*. Aug. 6, 2015. url: http://www.w3cook.com/os/summary/ (visited on 08/06/2015).

[Wal20] Dan Walsh. *An introduction to crun, a fast and low-memory footprint container runtime*. Publisher: Red Hat, Inc. Section: Enable Sysadmin. Aug. 3, 2020. url: https://www.redhat.com/sysadmin/introduction-crun (visited on 08/01/2024).

[Win24] Wind River Systems, Inc. *VxWorks: Das führende RTOS für den Intelligent Edge*. 2024. url: https://www.windriver.com/products/vxworks-de (visited on 06/07/2024).

[you24a] youki contributers. *youki*. original-date: 2021-03-27T11:06:58Z. July 9, 2024. url: https://github.com/containers/youki (visited on 07/09/2024).

[you24b] youki contributers. *youki/README.md*. GitHub. June 28, 2024. url: https://github.com/containers/youki/blob/e8b2af9b841d5bb20f0c341cad6f6e6cd4c25/README.md (visited on 08/14/2024).

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum github.com/m4schini